ROBERT SEDGEWICK

Cẩm nang

Thuật Toán

NHÀ XUẤT BẢN KHOA HỌC VÀ KỸ THUẬT

ROBERT SEDGEWICK

CẨM NANG
THUẬT TOÁN

Tập 1 : CÁC THUẬT TOÁN THÔNG DỤNG

(In lần thứ 5)

Người dịch : TRẦN ĐAN THƯ
VŨ MẠNH TƯỜNG
DƯƠNG VŨ DIỆU TRÀ
NGUYỄN TIẾN HUY

Hiệu đính : Gs.Ts. HOÀNG KIẾM

NHÀ XUẤT BẢN KHOA HỌC VÀ KỸ THUẬT

Algorithms

Robert Sedgewick, Princeton University (USA),
2nd Edition
Addison-Wesley Publishing Co.

LỜI NÓI ĐẦU

Trước khi viết một chương trình cho máy tính cho dù đơn giản nhất, bất cứ ai, dù ở trình độ nào, cũng đều phải suy tư ít nhiều về THUẬT TOÁN. Trong tổng thể kiến thức về TIN HỌC, các Thuật toán (Algorithms) cùng Cấu trúc dữ liệu (Data Structures) được xem là những tri thức quan trọng hàng đầu và không thể thiếu cho bất kỳ người lập trình (ứng dụng hay hệ thống) nào muốn đạt mục đích với hiệu quả cao nhất.

Tuy nhiên, cho đến nay, trong hầu hết các ấn phẩm và giáo trình Tin học, các thuật toán đều trình bày hoặc ở dạng quá "nôm na" hay rườm rà qua vài ví dụ đơn giản bằng lời hay các lưu đồ (flowcharts); hoặc lại quá trừu tượng khi dùng đến các khái niệm của lý thuyết và độ phức tạp thuật toán. Do đó, người học cũng như người lập trình đều cảm thấy thiếu các căn cứ tin cậy về các thuật toán - những "điểm tựa" vững để tạo ra thế giới các chương trình, phần mềm... như thế nhà bác học Archimède cổ xưa từng mơ ước dùng cho "đòn bẩy" nâng bổng cả Trái đất. Các khiếm khuyết đó đã được khắc phục trọn vẹn trong cuốn sách này. Nó được dịch trọn từ nguyên bản tiếng Anh cuốn "Algorithms" [của Robert Sedgewick, Princeton University (USA), Second Edition], do Addison-Wesley Publishing Co. xuất bản, tái bản nhiều lần; và nay đã trở thành một trong các tư liệu "kinh điển" cả về lý thuyết lẫn thực hành, cho người lập trình trên thế giới.

Mục tiêu chủ chốt của cuốn sách này là tổng hợp có hệ thống các phương pháp cơ bản, từ nhiều lãnh vực ứng dụng riêng biệt, nhằm cung cấp các giải thuật tốt nhất đã được kiểm chứng và công bố, để giải các bài toán cụ thể bằng máy tính.

Cuốn sách gồm 45 chương, chia thành 8 phần, tương ứng theo các phạm trù ứng dụng lớn. Các chương được trình bày theo trình tự từ căn bản đến cao cấp. Tuy nhiên, bạn đọc cũng có thể sử dụng từng phần tương đối độc lập, theo nhu cầu riêng của mình.

Cuốn sách được dịch và xuất bản thành hai tập :

- Tập I (tập này) : Các thuật toán thông dụng, gồm 23 chương, thuộc bốn phần đầu của nguyên bản;
- Tập II (sẽ xuất bản) : Các thuật toán chuyên dụng, gồm 22 chương, thuộc bốn phần cuối của nguyên bản.

[Để bạn đọc tiện theo dõi nội dung và tìm đọc, trong MỤC LỤC ở cuối tập I này, có in mục lục của cả cuốn sách, trọn 8 phần].

Cuốn sách được dùng cho các sinh viên ngành Tin học hoặc các đối tượng khác đã làm quen với máy tính và có kỹ năng nhất định về lập trình. Họ cần biết vận dụng một ngữ trình nào đó (ít nhất là Pascal chuẩn, vì các thuật toán ở đây được trình bày dưới dạng "tựa Pascal"). Sách cũng hữu ích cho các thày và trò các trường Trung học hệ đào tạo chuyên Tin.

Với tư cách một tài liệu tham khảo nghiêm túc, cuốn sách cũng hữu ích cho các cán bộ nghiên cứu ngành khác, hay cho những ai có nhu cầu phát triển các phần mềm hệ thống hoặc các trình ứng dụng. Bởi lẽ ngoài nội dung thuật toán, nó còn cung cấp chi tiết các thông tin hữu dụng về cài đặt và hiệu quả sử dụng chúng.

Mặc dù nhóm dịch thuật đã cố gắng bám sát nguyên bản, song không khỏi còn thiếu sót (nhất là về các thuật ngữ tiếng Việt còn nhiều bàn cãi); song chúng tôi và Nhà Xuất Bản vẫn hy vọng cuốn sách sẽ đáp ứng đúng nhu cầu và được đông đảo bạn đọc quan tâm đến Tin Học đón nhận, như một cuốn Cẩm Nang tra cứu.

TP.HCM, 30/4/1994
Gs.Ts. HOÀNG KIẾM
Cv.Ks. NGUYỄN PHÚC TRƯỜNG SINH

Thư từ liên hệ và góp ý xin gởi về :

- Nhà Xuất bản Khoa Học và Kỹ Thuật,
  70 Trần Hưng Đạo, Hà-Nội, ĐT - 014.2.54786.
- Chi nhánh NXB KH&KT,
  28 Đồng Khởi, Q.1, Thành phố HCM, ĐT - 225062.

1
GIỚI THIỆU

Mục tiêu của quyển sách này là nghiên cứu một loạt các thuật toán quan trọng và hữu ích : các phương pháp giải các bài toán thích hợp với việc cài đặt trên máy tính. Chúng ta sẽ đối diện với nhiều lĩnh vực áp dụng khác nhau, luôn cố gắng tập trung vào các thuật toán "cơ sở" quan trọng cần phải biết và thú vị để nghiên cứu. Do có một lượng lớn các lĩnh vực và thuật toán sẽ được giới thiệu, nên chúng ta sẽ không thể nghiên cứu nhiều phương pháp một cách thật chi tiết. Tuy nhiên, ta sẽ cố gắng bỏ ra đủ thời gian cho mỗi thuật toán để hiểu được các đặc trưng chủ yếu của thuật toán và chú ý đến các sự tinh tế của nó. Tóm lại, mục đích của chúng ta là học một lượng lớn các thuật toán quan trọng nhất được dùng trên các máy tính ngày nay, đủ tốt để có thể dùng được và hiểu rõ giá trị của chúng.

Để học một thuật toán tốt, ta phải cài đặt và chạy nó. Tương ứng với nó, một chiến lược được đề nghị để hiểu các chương trình sẽ trình bày trong quyển sách này là cài đặt và thử chúng, thí nghiệm với các biến thể, và thử chúng trên các bài toán thực tế. Chúng ta sẽ dùng ngôn ngữ lập trình Pascal để bàn luận và cài đặt hầu hết các thuật toán, tuy nhiên, do chúng ta chỉ sử dụng một tập con tương đối nhỏ của ngôn ngữ, nên các chương trình của chúng ta có thể dễ dàng được dịch thành nhiều ngôn ngữ lập trình hiện đại khác.

Các độc giả của quyển sách này cần có ít nhất một năm kinh nghiệm về lập trình trong các ngôn ngữ cấp cao và cấp thấp. Cũng vậy, các kiến thức về những cấu trúc dữ liệu đơn giản dùng trong các thuật toán cơ sở như mảng, ngăn xếp, hàng đợi và cây có thể là hữu ích, mặc dù những cấu trúc này sẽ được ôn lại một cách chi tiết trong chương 3 và 4. Một sự hiểu biết cơ sở về tổ chức máy, về ngôn ngữ lập trình và những khái niệm tin học cơ bản khác cũng

được yêu cầu (chúng ta sẽ ôn lại chúng một cách vắn tắt vào lúc thích hợp, nhưng luôn trong ngữ cảnh của việc giải các bài toán cụ thể). Một vài lĩnh vực áp dụng mà ta sẽ gặp cần tri thức về số học cơ sở. Ta cũng sẽ dùng một vài chất liệu toán học rất cơ bản như đại số tuyến tính, hình học, và toán rời rạc, nhưng không cần phải có trước các kiến thức về những chủ đề này.

THUẬT TOÁN

Khi viết một chương trình máy tính, ta thường cài đặt một phương pháp đã được nghĩ ra trước đó để giải quyết một vấn đề. Phương pháp này thường là độc lập với một máy tính cụ thể sẽ được dùng : nó hầu như thích hợp như nhau cho nhiều máy tính. Trong bất kỳ trường hợp nào, thì phương pháp, chứ không phải là bản thân chương trình máy tính là cái phải được nghiên cứu để học cách làm thế nào tấn công vào bài toán. Từ "Thuật toán" được dùng trong khoa học may tính để mô tả một phương pháp giải bài toán thích hợp cho việc cài đặt như là các chương trình máy tính. Thuật toán là "chất liệu" của khoa học máy tính : chúng là các đối tượng nghiên cứu trung tâm trong nhiều, nếu không nói là hầu hết, các lĩnh vực của Tin học.

Hầu hết các thuật toán đáng chú ý cần đến các phương pháp tổ chức dữ liệu phức tạp ám chỉ trong lúc tính toán. Các đối tượng được tạo ra theo cách này được gọi là các cấu trúc dữ liệu, và chúng cũng là các đối tượng trung tâm cần nghiên cứu trong khoa học máy tính. Vì vậy các thuật toán và cấu trúc dữ liệu đi liền với nhau; trong cuốn sách này chúng ta sử dụng quan điểm là cấu trúc dữ liệu tồn tại như các sản phẩm cuối cùng của các thuật toán. Các thuật toán đơn giản có thể phát sinh với các cấu trúc dữ liệu phức tạp và ngược lại, các thuật toán phức tạp có thể dùng các cấu trúc dữ liệu đơn giản. Ta sẽ nghiên cứu các tính chất chi tiết của nhiều cấu trúc dữ liệu trong cuốn sách này ; thực ra thì quyển sách này nên được gọi là "Thuật toán và cấu trúc dữ liệu".

Khi một chương trình máy tính rất lớn đang được phát triển, một nỗ lực lớn sẽ phải bỏ ra để hiểu được và định nghĩa được bài toán sẽ giải quyết, quản lý được độ phức tạp của nó, và phân rã nó thành các vấn đề nhỏ hơn mà nó có thể dễ dàng được cài đặt. Thực

tế là nhiều thuật toán được yêu cầu, sau khi phân rã lại trở thành tầm thường khi cài đặt. Tuy nhiên, trong hầu hết các trường hợp, có một vài thuật toán mà sự chọn lựa của nó là sống còn bởi vì hầu hết các tài nguyên hệ thống sẽ được tiêu thụ để chạy các thuật toán đó. Trong quyển sách này ta sẽ nghiên cứu một loạt các thuật toán cơ sở, nền tảng cho các chương trình lớn trong nhiều lĩnh vực áp dụng.

Việc dùng chung chương trình trong những hệ thống máy tính đang ngày càng trở nên phổ biến hơn, sao cho trong khi những người sử dụng máy tính chuyên nghiệp sẽ dùng đến phần lớn các thuật toán trong cuốn sách này, thì họ có thể chỉ cần cài đặt một phần nhỏ nào đó từ chúng mà thôi. Tuy nhiên, việc cài đặt các phiên bản đơn giản của các thuật toán cơ cở sẽ giúp chúng ta hiểu được chúng tốt hơn và vì vậy việc dùng các phiên bản nâng cao sẽ hiệu quả hơn. Cũng vậy, các cơ chế cho việc dùng chung phần mềm trên nhiều hệ thống máy tính thường khiến cho nó khó khăn khi phải cắt tỉa các chương trình chuẩn để thực hiện một cách hiệu quả trên các công việc đặc thù, khiến cho cơ hội để cài đặt lại các thuật toán cơ sở sẽ là xảy ra thường xuyên.

Các chương trình máy tính thường quá tối ưu. Có thể không bõ công khi chuốc lấy nỗi khổ phải bảo đảm rằng một cài đặt là hiệu quả nhất có thể trừ phi một thuật toán sẽ được dùng nhiều lần. Nếu không, thì chỉ cần một cài đặt tương đối đơn giản và cẩn thận là đủ : ta có thể có một sự tin tưởng nào đó rằng nó sẽ hoạt động, và có thể nó sẽ chạy chậm hơn năm hay mười lần phiên bản tốt nhất có thể có, điều đó có nghĩa là nó có thể chạy chậm hơn một vài giây. Trái lại, việc chọn một thuật toán thích hợp ngay từ đầu có thể tạo ra một sự cách biệt về tỉ lệ từ hàng trăm hay hàng ngàn hay hơn nữa, mà nó có thể chuyển sự chênh lệch này thành hàng nhiều phút, nhiều giờ hay hơn nữa về thời gian chạy. Trong cuốn sách này, chúng ta tập trung vào các cài đặt hợp lý đơn giản nhất của các thuật toán tốt nhất.

Thông thường nhiều thuật toán khác nhau (hay về cài đặt khác nhau) là khả dụng để giải quyết cùng một bài toán. Việc chọn lựa thuật toán tốt nhất cho một công việc cụ thể là một tiến trình rất phức tạp, thường cần đến việc phân tích toán học tinh vi. Chuyện

ngành Tin học nghiên cứu các vấn đề như vậy được gọi là Phân tích thuật toán. Nhiều thuật toán mà chúng ta sẽ nghiên cứu đã được chứng minh qua việc phân tích, là có hiệu năng rất tốt, trong khi các thuật toán khác đơn giản chỉ được biết là đã hoạt động tốt qua kinh nghiệm. Chúng ta sẽ không dừng lại ở các kết quả về hiệu năng đáng kể : mục đích của chúng ta là học một vài thuật toán hợp lý cho các công việc quan trọng. Nhưng ta không nên dùng một thuật toán mà không có một ý tưởng nào đó về những tài nguyên mà nó có thể tiêu thụ, như vậy, ta sẽ biết được các thuật toán của ta có thể được yêu cầu thực hiện như thế nào.

SƠ LƯỢC VỀ CÁC CHỦ ĐỀ

Dưới đây là các mô tả ngắn gọn về các phần chính của cuốn sách, cung cấp một vài chủ đề cụ thể được giới thiệu cũng như một dấu hiệu nào đó về khuynh hướng tổng quát của chúng ta để tiến tới các nội dung. Tập các chủ đề này dự định đề cập đến càng nhiều thuật toán cơ sở càng tốt. Một vài lĩnh vực được giới thiệu là các lĩnh vực tin học " cốt lõi " mà ta sẽ nghiên cứu chi tiết để học các thuật toán cơ bản có ứng dụng rộng rãi. Những lĩnh vực khác là các lĩnh vực nghiên cứu cao cấp trong Tin học và những lĩnh vực có liên quan như giải tích số, các phép toán tìm kiếm, kiến tạo trình biên dịch, và lý thuyết thuật toán - trong các trường hợp này sự bàn luận của chúng ta sẽ được dùng như một giới thiệu về các lĩnh vực này qua việc khảo sát một vài phương pháp cơ bản.

PHẦN CƠ SỞ trong ngữ cảnh của quyển sách này là các công cụ và phương pháp được dùng xuyên suốt cho các chương sau. Nó gồm một bàn luận ngắn về Pascal, theo sau là một giới thiệu về các cấu trúc dữ liệu cơ bản, gồm mảng, xâu liên kết, ngăn xếp, hàng đợi và cây. Chúng ta sẽ bàn về công dụng thực tiễn của đệ quy, và giới thiệu cách tiếp cận cơ bản của chúng ta hướng tới việc phân tích và cài đặt các thuật toán.

SẮP XẾP : Các phương pháp để sắp xếp lại các tập tin theo thứ tự là có tầm quan trọng cơ bản và được giới thiệu chi tiết. Một loạt các phương pháp sẽ được phát triển, được mô tả và được so sánh với nhau. Các thuật toán cho nhiều vấn đề có liên quan sẽ được xem xét, gồm có hàng đợi ưu tiên, phép chọn và phép trộn.

Một vài thuật toán trong số này được dùng như là nền tảng cho các thuật toán khác tiếp sau trong quyển sách.

TÌM KIẾM : Các phương pháp để tìm các vật trong các tập tin thì cũng có tầm quan trọng cơ bản. Chúng ta sẽ bàn luận về các phương pháp cơ bản và nâng cao để tìm kiếm bằng cách dùng cây và các phép biến đổi khóa số, kể cả các cây tìm kiếm nhị phân, cây cân bằng, phép băm, cây tìm kiếm số và trie, và các phương pháp thích hợp cho các tập tin rất lớn. Các mối quan hệ giữa những phương pháp sắp xếp cũng được chỉ ra.

Các thuật toán **XỬ LÝ CHUỖI** gồm một loạt các phương pháp để phân tích câu. Các kỹ thuật nén tập và mật mã cũng sẽ được khảo sát. Cũng vậy, một giới thiệu về các chủ đề nâng cao cũng được cung cấp, qua việc xem xét một vài bài toán cơ bản quan trọng trong phạm vi của chúng.

Các thuật toán **HÌNH HỌC** là một sự tập hợp có chọn lọc các phương pháp để giải quyết các bài toán liên quan đến điểm và đường (và các đối tượng hình học đơn giản khác) mà chỉ gần đây mới trở nên thông dụng. Chúng ta sẽ xem xét các thuật toán để tìm bao lồi của một tập các điểm, để tìm các phần giao giữa các đối tượng hình học, để giải các bài toán điểm gần nhất, và để tìm kiếm nhiều chiều. Có nhiều phương pháp trong số này hỗ trợ tốt cho các phương pháp sắp xếp và tìm kiếm cơ bản khác.

Các thuật toán **ĐỒ THỊ** hữu ích để giải một loạt các bài toán khó và quan trọng. Một chiến lược tổng quát để tìm kiếm trên các đồ thị sẽ được phát triển và được áp dụng cho các bài toán liên thông cơ bản, gồm có đường đi ngắn nhất, cây liên thông tối thiểu, mạng, và so khớp. Một sự xem xét thống nhất đối với các thuật toán này chứng tỏ rằng tất cả đều dựa trên cùng một thủ tục, và thủ tục này phụ thuộc vào một cấu trúc dữ liệu cơ bản đã được phát triển trước đó.

CÁC THUẬT TOÁN TOÁN HỌC gồm các phương pháp cơ bản từ số học và giải tích số. Chúng ta sẽ nghiên cứu các phương pháp liên quan với số học các số nguyên, đa thức, và ma trận cũng như các thuật toán để giải một loạt các vấn đề toán học mà nó phát sinh trong nhiều ngữ cảnh : việc tạo số ngẫu nhiên, lời giải của các

phương trình đồng thời, xấp xỉ dữ liệu, và lấy tích phân. Sự nhấn mạnh thiên về các khía cạnh thuật toán của phương pháp, chứ không phải trên nền tảng toán học.

CÁC CHỦ ĐỀ CAO CẤP được thảo luận nhằm mục đích liên hệ các chất liệu trong cuốn sách với nhiều lĩnh vực nghiên cứu cao cấp khác. Phần cứng chuyên dụng, quy hoạch động, quy hoạch tuyến tính, tìm kiếm vét cạn, và vấn đề NP- đầy đủ sẽ được xem xét từ một quan điểm cơ bản để cho độc giả có một sự đánh giá nào đó đối với các lĩnh vực nghiên cứu cao cấp đáng chú ý đã được gợi ra bởi các vấn đề cơ bản được bắt gặp trong cuốn sách này.

Việc nghiên cứu các thuật toán là thú vị vì nó là một lĩnh vực mới (hầu như tất cả các thuật toán ta sẽ nghiên cứu là có từ 25 năm trở lại đây) với một truyền thống phong phú (một vài thuật toán đã được biết từ hàng ngàn năm trước đây). Những khám phá mới đang được tạo ra hàng ngày, và một ít thuật toán là được hiểu hoàn toàn. Trong quyển sách này chúng ta sẽ xem xét các thuật toán rắc rối, phức tạp và khó cũng như các thuật toán đẹp, đơn giản và dễ. Mục tiêu của chúng ta là hiểu được các thuật toán thuộc nhóm trước và đánh giá đúng các thuật toán thuộc nhóm sau trong ngữ cảnh của nhiều ứng dụng khác nhau có thể có. Trong khi làm như vậy, chúng ta sẽ khảo sát một loạt các công cụ hữu ích và phát triển một phương pháp "tư duy thuật toán" mà nó sẽ hỗ trợ tốt cho chúng ta trong các thử thách tính toán sắp tới.

2
PASCAL

Ngôn ngữ lập trình được dùng xuyên suốt cuốn sách này là Pascal. Mọi ngôn ngữ đều có những điểm hay và dở của nó, và vì vậy việc chọn lựa bất kỳ một ngôn ngữ nào cho một cuốn sách giống cuốn này sẽ có những ưu điểm và nhược điểm. Nhưng nhiều ngôn ngữ lập trình hiện đại thì tương tự nhau, do đó bằng cách dùng tương đối ít các cấu trúc ngôn ngữ và tránh các quyết định cài đặt dựa trên các đặc tính riêng của Pascal, chúng ta sẽ phát triển được các chương trình mà mó dễ dàng có thể dịch thành các ngôn ngữ khác. Mục đích của chúng tôi là trình bày các thuật toán theo một dạng đơn giản và trực tiếp nhất có thể; Pascal cho phép chúng ta làm được điều này.

Các thuật toán thường được diễn tả trong các giáo trình và các báo cáo nghiên cứu bằng những ngôn ngữ tưởng tượng - không may là điều này thường làm mờ đi các chi tiết và làm cho độc giả rất khó có một cài đặt hữu dụng. Trong quyển sách này chúng tôi có quan điểm là : một trong các cách tốt nhất để hiểu một thuật toán và để làm cho việc sử dụng chúng trở nên có giá trị là trải qua kinh nghiệm bằng một cài đặt thực sự. Pascal có đặc tính là đang được dùng rộng rãi, và độc giả được khuyến khích để trở thành quen thuộc với một môi trường lập trình Pascal cục bộ. Các cài đặt Pascal trong quyển sách này là các chương trình chạy được mà nó dự định sẽ được chạy, được thí nghiệm, được sửa đổi và được sử dụng.

Lợi điểm của việc dùng Pascal là nó được dùng rộng rãi và có những đặc trưng cơ bản tương tự như các ngôn ngữ lập trình hiện đại khác, bất lợi là nó thiếu nhiều đặc tính cần cho các thuật toán tinh vi (và có sẵn trong các môi trường lập trình khác). Một vài chương trình mà ta sẽ gặp có thể được đơn giản hóa bằng cách dùng các đặc trưng ngôn ngữ cao cấp hơn (một số không có trong

Pascal), nhưng điều này thực sự thường ít xảy hơn là người ta có thể nghĩ. Khi thích hợp, việc bàn luận về các chương trình như vậy sẽ giới thiệu các kết quả ngôn ngữ thích hợp.

Một mô tả súc tích của ngôn ngữ Pascal được cho trong quyển "Pascal User Manual and Report" của Wirth và Jensen mà nó dùng như một định nghĩa cho ngôn ngữ. Mục đích của chúng ta trong chương này không phải là nhắc lại các thông tin từ cuốn sách đó nhưng là để nghiên cứu việc cài đặt của một thuật toán đơn giản (nhưng cổ điển) mà nó minh họa một vài đặc trưng cơ bản của ngôn ngữ và kiểu thức mà chúng ta sẽ sử dụng.

Ví dụ : Thuật toán Euclid

Để bắt đầu, chúng ta sẽ xét một chương trình Pascal để giải một bài toán cơ bản cổ điển : "Tối giản một phân số cho trước". Chúng ta muốn viết 2/3, chứ không phải 4/6, 200/300 hay 178468/267702. Giải bài toán này tương đương với việc tìm ước số chung lớn nhất (USCLN) của tử số và mẫu số : số nguyên lớn nhất chia hết cả hai. Một phân số được tối giản bằng cách chia cả tử lẫn mẫu cho USCLN của chúng. Một phương pháp hiệu quả để tìm USCLN đã được khám phá bởi người Hy Lạp cổ đại trên 2000 năm trước : nó được gọi là thuật toán Euclid vì nó được giải thích chi tiết trong luận án nổi tiếng của Euclid là "Elements".

Phương pháp của Euclid dựa trên sự kiện là nếu u lớn hơn v thì USCLN của u và v bằng với USCLN của v và u-v. Nhận xét này dẫn tới cài đặt bằng Pascal ở trang sau.

Trước tiên, chúng ta xem xét các tính chất của ngôn ngữ được đưa ra bởi chương trình này. Pascal có một cú pháp cấp cao chặt chẽ mà nó cho phép định danh dễ dàng các đặc trưng chính của chương trình. Các biến (var) và các hàm (function) được dùng bởi chương trình thì được khai báo trước tiên, theo sau là thân của chương trình (các phần chương trình chính khác không được dùng trong chương trình trên, mà nó cũng được khai báo trước thân chương trình, là constants và types). Các hàm có cùng dạng như chương trình chính ngoại trừ chúng trả về một giá trị, mà nó được đặt lên bằng cách gán một cái gì đó vào tên hàm trong thân của hàm (**function** là một kiểu "thủ tục con" trong Pascal - kiểu còn

```
program Euclid (input, output) ;
var x, y : integer ;
function USCLN (u, v : integer) : integer ;
var t : integer ;
begin
   repeat
      if u<v then
         begin t := u ; u := v ; v := t
         end ;
      u := u-v
   until u = 0 ;
   USCLN := v
end ;
begin
   while not eof do
      begin readln(x,y) ;
         if (x>0) and (y>0) then writeln(x, y, USCLN (x, y))
      end ;
end.
```

lại, mà nó không có giá trị trả về là **procedure** (thủ tục)). Hàm có sẵn *readln* đọc một dòng từ ngõ nhập *(input)* và gán giá trị tìm thấy vào các biến đã được cho như các tham số ; *writeln* thì tương tự. Một tân từ chuẩn có sẵn, *eof*, được đặt là "đúng" (true) khi không còn dữ liệu nhập nào khác (việc nhập và xuất trong một dòng có thể dùng được với *read, write,* và *eoln*). Việc khai báo của input và output trong câu lệnh chương trình chỉ ra rằng chương trình đang sử dụng các dòng nhập và xuất "chuẩn".

Thân của chương trình trên thì tầm thường : nó đọc các cặp số từ ngõ nhập, sau đó, nếu cả hai đều dương thì ghi chúng và USCLN của chúng ra ngõ xuất (điều gì sẽ xảy ra nếu hàm USCLN được gọi với u và v là âm hay bằng 0 ?).

Hàm USCLN cài đặt chính thuật toán của Euclid : chương trình là một vòng lặp mà nó đầu tiên bảo đảm là u >= v bằng cách tráo đổi chúng, nếu cần, và sau đó thay u bằng u-v. USCLN của các biến u và v thì luôn luôn giống với USCLN của các giá trị gốc được đưa cho thủ tục : cuối cùng tiến trình kết thúc với u = v, cả hai cùng bằng với với USCLN của các giá trị ban đầu (và tất cả các giá trị trung gian) của u và v.

Thí dụ này được viết như một chương trình Pascal hoàn chỉnh mà độc giả có thể muốn dùng để trở thành quen với hệ thống lập trình Pascal nào đó. Phần "điều khiển" của chương trình đọc vào các cặp số, sau đó ghi chúng ra cùng với USCLN của chúng. Thí dụ này được đưa vào để minh họa làm thế nào để thực hành thuật toán, và để nhấn mạnh một điểm là các thuật toán trong cuốn sách này sẽ được hiểu tốt nhất khi chúng được cài đặt và được chạy trên một vài giá trị nhập mẫu. Tùy vào chất lượng của môi trường gỡ lỗi có sẵn, độc giả có thể mong muốn trang bị thêm cho các chương trình. Ví dụ như các giá trị trung gian được lấy bởi u và v trong vòng lặp **repeat** là đáng quan tâm trong chương trình ở trên chẳng hạn.

Mặc dù chủ đề của chúng ta trong phần này là ngôn ngữ, không phải là thuật toán, chúng ta phải biết đánh giá đúng thuật toán cổ điển của Euclid : cài đặt ở trên có thể được nâng cấp bằng cách chú ý rằng khi u>=v, ta tiếp tục trừ bớt u một bội số của v cho tới khi gặp một số nhỏ hơn u. Nhưng số này chính là phần dư còn lại sau khi chia u cho v, mà nó là giá trị hàm mod tính được: USCLN của u và v chính là USCLN của u và u mod v. Ví dụ, USCLN của 461952 và 116298 là 18, như được chỉ ra bởi dãy 461952, 116298, 113058, 3240, 2898, 342, 162, 18. (Mỗi số trong dãy này là phần dư còn lại sau khi chia hai số đứng trước cho nhau). Độc giả có thể muốn sửa đổi cài đặt ở trên để dùng toán tử mod và lưu ý xem sửa đổi đó hiệu quả hơn bao nhiêu khi, ví dụ như, tìm USCLN của một số rất lớn và một số rất nhỏ. Hóa ra thuật toán này luôn sử dụng một số bước tương đối nhỏ.

KIỂU DỮ LIỆU

Hầu hết các thuật toán trong quyển sách này thao tác trên các kiểu dữ liệu đơn giản : số nguyên, số thực, các ký tự, hay chuỗi ký tự. Một trong những đặc trưng quan trọng nhất của Pascal là khả năng tạo nên các kiểu dữ liệu phức tạp hơn từ những khối xây dựng cơ sở này. Đây là một trong những đặc trưng "cao cấp" mà ta tránh dùng, để giữ cho các ví dụ của chúng ta đơn giản và giữ cho trọng tâm của chúng ta tập trung vào tính cơ động của các thuật toán thay vì các tính chất của dữ liệu của chúng. Chúng ta tranh đấu để đạt được điều này mà không làm mất đi tính khái

quát. Thật vậy, tính rất dễ dùng của các đặc trưng cao cấp như của Pascal cung cấp sẽ khiến cho ta dễ dàng chuyển một thuật toán từ một "thứ đồ chơi" mà nó thao tác trên các kiểu dữ liệu đơn giản thành một "thứ ngựa thồ" mà nó thao tác trên các mẩu tin (record) phức tạp. Khi các phương pháp cơ sở được mô tả tốt nhất bằng thuật ngữ của các kiểu do người dùng định nghĩa, thì ta sẽ làm như vậy. Ví dụ, các phương pháp hình học trong các chương 24 đến 28 là dựa trên các kiểu dành cho điểm, đoạn thẳng, đa giác, v.v...

Đôi khi có trường hợp là biểu diễn cấp thấp thích hợp của dữ liệu lại là chìa khóa cho hiệu suất. Một cách lý tưởng, là phương pháp mà một chương trình thực hiện không nên phụ thuộc vào việc là các số sẽ được biểu diễn như thế nào hay các ký tự được nén như thế nào (nhặt ra hai ví dụ), nhưng cái giá mà ta phải trả về hiệu suất qua việc theo đuổi ý tưởng này thường là quá cao. Các lập trình viên trong quá khứ đã đáp ứng trường hợp này bằng cách thực hiện một bước nhảy mạnh mẽ tới hợp ngữ hay ngôn ngữ máy, ở đó có ít ràng buộc về biểu diễn. May mắn thay, các ngôn ngữ cấp cao hiện đại cung cấp các cơ chế để tạo ra các biểu diễn "nhạy bén" mà không phải đi tới các thái cực như vậy. Điều này cho phép chúng ta thưởng thức được một vài thuật toán cổ điển quan trọng. Dĩ nhiên, các cơ chế như vậy nhất thiết là phụ thuộc máy, và chúng ta sẽ không xem xét chúng quá chi chi tiết, ngoại trừ là để chỉ ra khi nào thì chúng thích hợp. Kết quả này được bàn luận chi tiết hơn trong các chương 10, 17 và 22, trong đó các thuật toán dựa trên các biểu diễn nhị phân của dữ liệu được xem xét.

Ta cũng cố gắng tránh đối phó với các kết quả biểu diễn phụ thuộc máy khi xem xét các thuật toán mà nó thao tác trên các ký tự và chuỗi ký tự. Thông thường, chúng ta đơn giản hóa các ví dụ bằng cách chỉ làm việc với các ký tự in hoa từ A tới Z, dùng một mã đơn giản với ký tự thứ i của bảng chữ cái được biểu diễn bởi số nguyên i. Biểu diễn của các ký tự và các chuỗi ký tự là một phần cơ bản trong giao tiếp giữa người lập trình, ngôn ngữ lập trình và máy, mà người ta nên bảo đảm là hiểu nó đầy đủ trước khi cài đặt các thuật toán để xử lý các dữ liệu như vậy. Các phương pháp đã cho trong quyển sách này dựa trên các biểu diễn đã được đơn giản hóa thì dễ dàng được thích ứng.

Chúng ta dùng số nguyên (integer) bất cứ khi nào có thể. Các chương trình mà nó xử lý các số thực (real) rơi vào lĩnh vực của giải tích số. Cụ thể là hiệu năng của chúng bị ràng buộc mật thiết vào các tính chất toán học của biểu diễn. Chúng ta sẽ trở lại vấn đề này trong các chương 37, 38, 39, 41 và 43, trong đó có một vài thuật toán số cơ bản được bàn đến. Tạm thời, chúng ta bám vào số nguyên ngay cả khi số thực có thể là thích hợp hơn, để tránh sự không hiệu quả và không chính xác thường gắn liền với các biểu diễn "dấu phẩy động".

NHẬP/XUẤT

Một lĩnh vực khác có sự phụ thuộc máy đáng kể là tương tác giữa chương trình và dữ liệu của nó, thường được xem như nhập-xuất. Trong các hệ điều hành, thuật ngữ này có liên quan tới việc chuyển dữ liệu giữa máy tính và môi trường vật lý như băng từ hay đĩa : chúng ta sẽ đụng đến các chủ đề đó chỉ trong các chương 13 và 18. Thông thường nhất là chúng ta chỉ đơn giản tìm một cách có hệ thống để nhận dữ liệu và phát sinh các kết quả từ các cài đặt của các thuật toán, ví dụ như chương trình USCLN ở trên.

Khi "đọc" và "ghi" được thực hiện, chúng ta sẽ dùng các đặc trưng chuẩn của Pascal nhưng cố gắng sử dụng càng ít càng tốt các phương tiện định dạng bổ sung. Một lần nữa, việc làm này là để giữ cho các chương trình súc tích, dễ mang chuyển và dễ chuyển đổi : một cách trong đó độc giả có thể mong muốn sửa đổi các chương trình là thêm thắt các giao tiếp của chúng với lập trình viên. Một vài môi trường lập trình Pascal hay các môi trường hiện đại khác thực sự dùng read hay write để liên hệ đến một môi trường ngoài : thực sự là chúng thường liên hệ tới các "thiết bị luận lý" hay các "dòng" dữ liệu. Vì vậy, kết xuất của chương trình có thể được dùng như dữ liệu nhập cho một chương trình khác, mà không cần bất kỳ một phép đọc hay viết vật lý nào. Xu hướng biến thành dòng việc xử lý nhập/xuất trong các cài đặt của chúng ta sẽ khiến cho chúng trở nên hữu ích hơn trong các môi trường đó.

Thực sự, trong nhiều môi trường lập trình hiện đại, thật thích

hợp và khá dễ dàng dùng các biểu diễn hình ảnh như những cái đã được dùng trong các hình vẽ xuyên suốt cuốn sách (như đã được mô tả trong Epilog, các hình ảnh này đã thực sự được sinh ra bởi chính các chương trình đó, với một giao diện đã được thêm thắt một cách rất có ý nghĩa).

Nhiều phương pháp ta sẽ bàn đến được dự định dùng trong các hệ thống ứng dụng lớn hơn, vì vậy một cách thích hợp hơn cho chúng để nhận được dữ liệu của mình là qua các tham số. Đây là phương pháp được dùng cho thủ tục USCLN ở trên. Cũng vậy, nhiều cài đặt trong các chương sau này của cuốn sách dùng đến các chương trình từ những chương trước đó. Một lần nữa, để tránh làm chệch hướng chú ý của chúng ta khỏi chính các thuật toán, chúng ta chống lại sự cám dỗ là "đóng gói" các cài đặt cho việc sử dụng chúng như là các chương trình tiện ích tổng quát. Rõ ràng là nhiều cài đặt mà chúng ta nghiên cứu là hoàn toàn thích hợp như một điểm khởi đầu cho các trình tiện ích như vậy, nhưng một số lượng lớn các vấn đề về phụ thuộc hệ thống và phụ thuộc ứng dụng mà chúng ta bỏ qua ở đây sẽ phải được giải quyết một cách thấu đáo khi phát triển những gói chương trình như vậy.

Thông thường chúng ta viết các thuật toán thao tác trên các dữ liệu "toàn cục", để tránh việc truyền tham số quá nhiều. Ví dụ như hàm USCLN có thể thao tác trực tiếp trên x và y, thay vì dùng các tham số u và v. Đó không phải là một sự biện hộ trong trường hợp này vì USCLN là một hàm được định nghĩa tốt với hai cái nhập của nó. Tuy nhiên, khi nhiều thuật toán thao tác trên cùng dữ liệu, hay khi có một số lượng lớn dữ liệu được truyền đi, thì chúng ta sẽ dùng các biến toàn cục để tiết kiệm trong việc diễn đạt các thuật toán và để tránh việc chuyển các dữ liệu một cách không cần thiết. Các đặc trưng cấp cao có trong Pascal và các ngôn ngữ cũng như các hệ thống khác sẽ cho phép điều này được thực hiện một cách sạch sẽ hơn, nhưng, một lần nữa, khuynh hướng của chúng ta là tránh các sự phụ thuộc ngôn ngữ như vậy khi có thể được.

CÁC LƯU Ý KẾT THÚC

Nhiều ví dụ khác tương tự chương trình Euclid ở trên đã được cho trong "Pascal User Manual and Report" và trong các chương sau. Độc giả được khuyến khích đò trong "manual", cài đặt và kiểm tra một số chương trình đơn giản và sau đó đọc "manual" một cách cẩn thận để trở nên thật quen với hầu hết các đặc trưng của Pascal.

Các chương trình Pascal được cho trong cuốn sách này dự định sẽ phục vụ như là các mô tả chính xác của các thuật toán, như là các ví dụ về các cài đặt trọn vẹn, và như là điểm khởi đầu cho các chương trình thực tế. Như đã lưu ý ở trên, các độc giả đã quen dùng các ngôn ngữ khác sẽ gặp một ít khó khăn khi đọc các thuật toán được biểu diễn bằng Pascal và sau đó, khi cài đặt chúng trong một ngôn ngữ khác. Ví dụ, chương trình sau là một cài đặt của thuật toán Euclid trong C.

```
# include <stdio.h>
main ()
{
    int x, y ;
    while (scanf ("%d %d", & x, & y) ! = EOF)
        if ((x>=0) && (y>=0))
            printf ("%d %d %d\n", x, y, USCLN (x,y));
}
int USCLN (u,v)
    int u, v ;
    {
    int t ;
    do {
    if (u)
        {t = u ; u = v ; v = t ; } ;
    u = u-v ;
    }while (u! = v) ;
return (u) ;
}
```

Đối với thuật toán này, gần như có một sự tương ứng 1-1 giữa các lệnh của Pascal và C, như dự kiến, mặc dù có những cài đặt súc tính hơn trong cả hai ngôn ngữ.

BÀI TẬP

1. Cài đặt phiên bản cổ điển của thuật toán Euclid như đã mô tả trong tài liệu.
2. Kiểm tra xem các giá trị nào mà hệ thống Pascal của bạn tính được cho u mod v khi u và v không nhất thiết là dương.
3. Cài đặt một thủ tục để tối giản một phân số cho trước, bằng cách dùng một kiểu
   type fraction = record numerator, denominator: integer end;
4. Viết một hàm function convert : integer mà nó đọc vào một số thập phân từng ký tự (số) ở mỗi thời điểm, kết thúc bởi khoảng trống, và trả về giá trị của số đó.
5. Viết một thủ tục procedure binary (x : integer) mà nó in ra dạng nhị phân tương đương của một số.
6. Hãy cho tất cả các giá trị mà u và v lấy khi USCLN được gọi với lệnh gọi khởi đầu là USCLN (12345, 56789).
7. Chính xác có bao nhiêu lệnh Pascal được thi hành cho lệnh gọi trong bài tập trước ?
8. Viết chương trình để tính USCLN của ba số nguyên u, v và w.
9. Tìm cặp số lớn nhất có thể biểu diễn được như các số nguyên trong hệ thống Pascal của bạn mà USCLN của chúng là 1.
10. Cài đặt thuật toán Euclid bằng FORTRAN hay BASIC.

3
CÁC CẤU TRÚC DỮ LIỆU CƠ BẢN

Trong chương trình này, chúng ta sẽ thảo luận các phương pháp cơ bản của việc tổ chức dữ liệu dùng cho việc xử lý bởi các chương trình máy tính. Đối với nhiều ứng dụng, việc chọn lựa cấu trúc dữ liệu (CTDL) thích hợp sẽ thực sự là quyết định duy nhất quan trọng được ngầm hiểu trong việc cài đặt : một khi chọn lựa đã được tạo ra, thì chỉ cần đến những thuật toán rất đơn giản. Đối với cùng một dữ liệu, một vài CTDL cần đến nhiều hay ít chỗ hơn những cái khác ; đối với cùng các thao tác trên dữ liệu, một vài CTDL dẫn tới các thuật toán hiệu quả hơn hay kém so với những cái khác. Chủ đề này sẽ thường lặp lại xuyên suốt quyển sách này, vì việc chọn lựa thuật toán và CTDL được quyện chặt vào nhau, và chúng ta tiếp tục tìm kiếm các phương pháp để tiết kiệm thời gian hay không gian bằng cách tạo ra chọn lựa này một cách thích đáng.

Một CTDL không phải là một đối tượng thụ động : chúng ta cũng phải xem xét các thao tác sẽ được thực hiện trên nó (và những thuật toán được dùng cho các thao tác này). Khái niệm này được chuẩn hóa trong khái niệm của một kiểu dữ liệu trừu tượng mà chúng ta sẽ bàn ở cuối chương này. Nhưng mối quan tâm chủ yếu của chúng ta là các cài đặt cụ thể, và chúng ta sẽ tập trung vào các biểu diễn và thao tác cụ thể.

Chúng ta sẽ xem xét mảng, xâu liên kết, ngăn xếp, hàng đợi, và các biến thể đơn giản khác. Đây là các CTDL cổ điển với tính ứng dụng rộng rãi : cùng với cây (xem chương 4), chúng tạo thành nền tảng cho hầu như tất cả các thuật toán được xét trong cuốn sách này. Trong chương này, chúng ta xem xét các biểu diễn cơ sở và các phương pháp cơ bản để sử dụng các cấu trúc này, thực hiện

qua một số ví dụ đặc thù về tính ứng dụng của chúng, và thảo luận các chủ đề có liên quan như quản lý bộ nhớ.

MẢNG (array)

Có lẽ CTDL cơ bản nhất là mảng, được định nghĩa như là cấu trúc nguyên sơ trong Pascal và hầu hết các ngôn ngữ lập trình khác. Một mảng là một số cố định các mẫu dữ liệu được chứa một cách liên tục và có thể được truy xuất bởi một chỉ mục (index). Chúng ta coi phần tử thứ i của một mảng a như là a[i]. Trách nhiệm của người lập trình là chứa một cái gì đó có nghĩa trong vị trí mảng a[i] trước khi tham khảo tới nó; sao lãng điều này là một trong những lỗi lập trình phổ biến nhất.

Một ví dụ đơn giản của việc dùng một mảng cho bởi chương trình sau, mà nó in ra tất cả các số nguyên tố nhỏ hơn 1000. Phương pháp được sử dụng, ra đời từ thế kỷ thứ 3 trước Công nguyên, được gọi là "sàng Eratosthenes" :

```
program primes (input, output);
const N = 1000;
var   a: array [1..N] of boolean;
      i, j: integer;
begin a[1]:= false;   for i:= 2 to N do a[i]:= true;
   for i:= 2 to N div 2 do
      for j:= 2 to N div i do a[i*j]:= false;
   for i:= 1 to N do
      if a[i] then write(i:4);
end.
```

Chương trình này sử dụng một mảng có kiểu các phần tử đơn giản nhất, là các giá trị luận lý. Mục đích của chương trình là đặt a[i] về "đúng" (true) nếu i là số nguyên tố, "sai" (false) nếu không. Nó làm điều này bằng cách, với mỗi i, đặt phần tử mảng tương ứng với một bội số của i là false vì bất kỳ một số nào mà là một bội của bất kỳ một số khác thì không thể là nguyên tố. Sau đó nó đi học theo mảng một lần nữa, in ra các số nguyên tố. (Chương trình này có thể được làm cho hiệu quả hơn bằng cách thêm lệnh kiểm tra if

a[i] then trước vòng lặp for cho j, vì nếu i không là nguyên tố, thì tất cả các phần tử mảng tương ứng với tất cả các bội của nó đã phải được đánh dấu rồi). Lưu ý là đầu tiên mảng được "khởi động" để chỉ ra rằng không có một số nào được biết sẽ là phi nguyên tố: thuật toán sẽ đặt về false các phần tử mảng tương ứng với các chỉ mục mà nó được biết là phi nguyên tố.

Sàng Eratosthenes là trường hợp đặc trưng trong số các thuật toán mà nó khai thác sự kiện là bất kỳ một phần tử nào trong một mảng cũng có thể được truy xuất một cách hiệu quả. Thuật toán cũng truy xuất các phần tử của mảng một cách tuần tự, cái này sau cái kia. Trong nhiều ứng dụng, thứ tự tuần tự là quan trọng ; trong các ứng dụng khác thứ tự tuần tự được sử dụng do nó cũng tốt ngang với bất kỳ một thứ tự nào khác. Nhưng đặc trưng chủ yếu của các mảng là, nếu chỉ mục đã được biết, thì bất kỳ một phần tử nào cũng đều có thể được truy xuất trong thời gian không đổi.

Kích thước của mảng phải được biết trước; trong Pascal, nó phải được biết vào lúc dịch. Để chạy chương trình ở trên với một giá trị khác của N, thì cần phải thay đổi hằng số N, sau đó biên dịch và thi hành lại. Trong một vài môi trường lập trình, có thể khai báo kích thước của một mảng vào lúc chạy (sao cho, ví dụ như người ta có thể gõ vào một giá trị của N, và sau đó chương trình sẽ trả lời với các số nguyên tố nhỏ hơn N mà không bỏ phí bộ nhớ do phải khai báo một mảng lớn bằng với bất kỳ một giá trị nào mà người sử dụng được phép gõ vào), nhưng tính chất cơ bản của mảng vẫn là có kích thước cố định và phải được biết trước khi chúng được sử dụng.

Mảng là các CTDL cơ bản trong đó chúng có một sự tương ứng trực tiếp với các hệ thống bộ nhớ trên hầu hết các máy tính. Để nhận được nội dung một từ (word) của bộ nhớ trong ngôn ngữ máy, chúng ta cung cấp một địa chỉ. Vì vậy, ta có thể nghĩ về toàn thể bộ nhớ như là một mảng, với các địa chỉ bộ nhớ tương ứng với các chỉ mục mảng. Hầu hết các bộ xử lý ngôn ngữ máy tính dịch các chương trình có mảng thành các chương trình ngôn ngữ máy khá hiệu quả mà nó truy xuất bộ nhớ một cách trực tiếp.

Một cách tương tự khác để cấu trúc các thông tin là dùng một bảng số hai chiều được tổ chức thành hàng và cột. Ví dụ, một bảng điểm của các sinh viên trong một học trình có thể có một hàng cho mỗi sinh viên, một cột cho mỗi điểm. Trên một máy tính, một bảng như vậy sẽ được biểu diễn như một mảng hai chiều với hai chỉ mục, một cho hàng và một cho cột. Các thuật toán khác nhau trên các cấu trúc như vậy là đơn giản; ví dụ, để tính điểm trung bình của một cột điểm, ta cộng tất cả các phần tử trong một cột và chia cho số hàng; để tính điểm trung bình của một sinh viên cụ thể trong học trình, ta cộng tất cả các phần tử trong một hàng rồi chia cho số cột. Các mảng hai chiều được dùng rộng rãi trong các ứng dụng loại này. Thực sự, trên một máy tính, thường là dễ dàng và khá đơn giản khi dùng nhiều hơn hai chiều : một giám học có thể dùng một chỉ mục thứ ba để giữ các bảng điểm sinh viên cho một dãy các năm học.

Mảng cũng tương ứng trực tiếp với các vector, một thuật ngữ toán học dùng cho một danh sách các đối tượng có chỉ mục. Tương tự, các mảng hai chiều tương ứng với các ma trận. Chúng ta sẽ nghiên cứu các thuật toán để xử lý các đối tượng toán học này trong các chương 36 và 37.

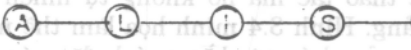
XÂU LIÊN KẾT (linked list)

CTDL cơ bản thứ hai cần xem xét là xâu liên kết, được định nghĩa như là một cấu trúc nguyên sơ trong một vài ngôn ngữ lập trình (như trong LISP) nhưng không phải là một cấu trúc nguyên sơ của Pascal. Tuy nhiên, Pascal cung cấp một vài thao tác nguyên sơ cơ bản mà nó khiến cho việc dùng các xâu liên kết trở nên dễ dàng.

Ưu điểm chủ yếu của xâu liên kết so với mảng là chúng có thể co giãn về kích thước trong thời gian sống của chúng. Cụ thể, kích thước tối đa của chúng không cần được biết trước. Trong các ứng dụng thực tế, điều này thường khiến cho có thể có nhiều CTDL chia xẻ cùng một không gian, mà không cần phải chú ý đặc biệt đến các kích thước tương đối của chúng ở bất kỳ một thời điểm nào.

Một ưu điểm thứ hai của xâu liên kết là chúng cung cấp tính linh hoạt trong việc cho phép các phần tử được tổ chức lại một cách hiệu quả. Tính mềm dẻo này có được là do tính truy xuất nhanh tới bất kỳ một phần tử tùy ý nào trong xâu. Điều này sẽ trở nên rõ ràng hơn dưới đây, sau khi chúng ta đã xem xét một vài tính chất cơ bản của xâu liên kết và một vài thao tác cơ sở chúng ta sẽ thực hiện trên chúng.

Một xâu liên kết là một tập các phần tử được tổ chức một cách tuần tự, giống như một mảng. Trong một mảng, sự tổ chức tuần tự được cung cấp ngầm (bởi vị trí trong mảng) ; trong một xâu liên kết, ta dùng một sự sắp xếp tường minh trong đó mỗi phần tử là một phần của một "nút" mà nó cũng có chứa một "liên kết" (link) tới nút kế. Hình 3.1 minh họa một xâu liên kết, với các phần tử được biểu diễn bởi các chữ cái, các nút bởi các cung tròn và các liên kết bởi các đoạn thẳng nối các nút. Chúng ta sẽ xem chi tiết dưới đây các nút được biểu diễn như thế nào trong máy tính ; hiện nay chúng ta sẽ chỉ nói đơn giản là các nút và các liên kết.



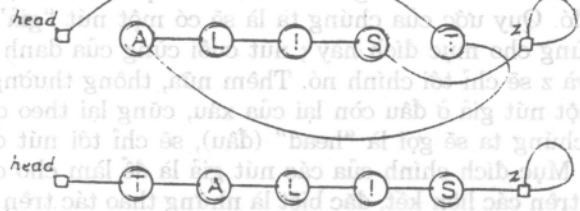
Hình 3.1 *Một xâu liên kết*

Ngay cả biểu diễn đơn giản ở hình 3.1 cũng phơi bày hai chi tiết mà ta phải xem xét. Đầu tiên là mỗi nút có một liên kết, như thế liên kết ở nút cuối cùng của danh sách phải chỉ ra một nút "kế" nào đó. Quy ước của chúng ta là sẽ có một nút "giả", mà ta gọi là z, dùng cho mục đích này ; nút cuối cùng của danh sách sẽ chỉ tới z, và z sẽ chỉ tới chính nó. Thêm nữa, thông thường chúng ta sẽ có một nút giả ở đầu còn lại của xâu, cũng lại theo quy ước. Nút này, chúng ta sẽ gọi là "head" (đầu), sẽ chỉ tới nút đầu tiên trong xâu. Mục đích chính của các nút giả là để làm cho các thao tác cụ thể trên các liên kết, đặc biệt là những thao tác trên các nút đầu và cuối của xâu được thuận tiện hơn. Các quy ước khác sẽ được bàn dưới dây. Hình 3.2 minh họa cấu trúc xâu, có thêm các nút giả này.

Hình 3.2 *Một xâu liên kết với các nút giả của nó*

Bây giờ, biểu diễn theo thứ tự tường minh này cho phép các thao tác sẽ dược thực hiện hiệu quả hơn nhiều so với mảng. Ví dụ, giả sử ta muốn chuyển T từ cuối danh sách ra đầu. Trong một mảng, ta phải chuyển mọi phần tử để có chỗ cho phần tử mới ở đầu; trong một xâu liên kết, ta chỉ thay đổi ba liên kết, như minh họa trong hình 3.3. Hai phiên bản được minh họa trong hình 3.3 là tương đương ; chỉ có điều là chúng được vẽ khác nhau. Chúng ta làm cho nút chứa T chỉ tới A, nút chứa S chỉ tới z, và "head" chỉ tới T. Cho dù nếu danh sách này có rất dài đi chăng nữa, ta vẫn có thể tạo ra sự thay đổi về cấu trúc này chỉ bằng cách thay đổi ba liên kết.

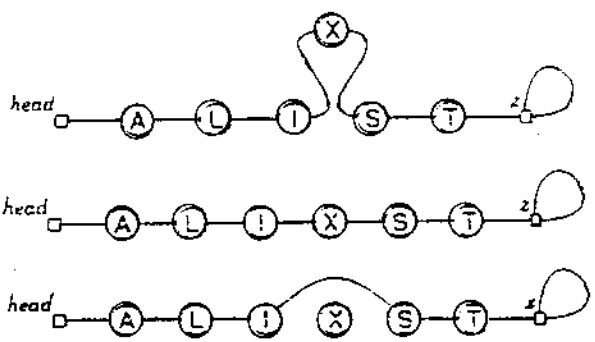
Quan trọng hơn, ta có thể nói đến việc "chèn" (inserting) một phần tử vào trong một xâu liên kết (khiến cho xâu dài ra thêm một phần tử nữa), một thao tác mà nó không tự nhiên và không dễ dàng trong một mảng. Hình 3.4 minh họa làm thế nào để chèn X vào trong xâu ví dụ của chúng ta bằng cách đặt nó vào trong một nút mà nó chỉ tới S, sau đó làm cho nút chứa I chỉ tới nút mới. Đối với thao tác này, chỉ cần thay đổi hai mối liên kết, không có vấn đề là xâu đó dài như thế nào.



Hình 3.3 *Sắp xếp lại một xâu liên kết*

Tương tự, ta có thể nói đến việc "xóa" (deleting) một phần tử khỏi một xâu liên kết (khiến cho xâu ngắn bớt một phần tử). Ví dụ, xâu thứ ba trong hình 3.4 cho thấy làm thế nào để xóa X khỏi xâu thứ hai bằng cách đơn giản là làm cho nút chứa I chỉ tới S, bỏ qua X. Bây giờ nút chứa X vẫn còn tồn tại (thực tế nó vẫn chỉ tới S), và có lẽ sẽ được giải phóng theo một cách nào đó, vấn đề là bây giờ nó không còn ở trong xâu này nữa, và không thể được truy xuất bởi việc dò theo các mối liên kết bắt đầu từ "head". Ta sẽ trở lại vấn đề này dưới đây.

Tuy nhiên, có những thao tác khác mà đối với xâu liên kết là không thích hợp. Hiển nhiên nhất trong số này là "tìm phần tử thứ k" (tìm một phần tử, cho trước chỉ mục của nó) : trong một mảng điều này được thực hiện đơn giản bằng cách truy xuất a[k], nhưng trong một xâu ta phải duyệt qua k mối liên kết.



Hình 3.4 *Chèn vào và xóa khỏi một xâu liên kết*

Một thao tác khác mà nó không tự nhiên trên xâu liên kết là "tìm phần tử nằm trước phần tử đã cho". Nếu tất cả những thứ mà chúng ta có là liên kết tới T trong xâu mẫu này, thì cách duy nhất để tìm thấy liên kết trỏ tới S là bắt đầu từ đầu (head) và duyệt qua từng nút để tìm được nút mà nó trỏ tới T. Vì thực tế, thao tác này là cần thiết nếu ta muốn xóa được một nút cho trước ra khỏi một xâu liên kết : nếu không, làm thế nào ta tìm thấy được nút mà liên kết của nó phải được thay đổi ? Trong nhiều ứng dụng, ta có thể tránh bài toán này bằng cách thiết kế lại thao tác xóa cơ bản

thành "xóa nút kế tiếp". Một bài toán tương tự có thể tránh được đối với phép chèn bằng cách tạo một thao tác chèn cơ bản là "chèn một phần tử cho trước vào sau một nút cho trước" trong xâu.

Pascal cung cấp các thao tác nguyên sơ mà nó cho phép các xâu liên kết sẽ được cài đặt một cách trực tiếp. Đoạn chương trình sau đây là một cài đặt mẫu cho các chức năng cơ bản mà ta đã thảo luận cho tới giờ.

```
type link = ↑node;
   node = record key : integer ; next : link end ;
var head , z , t : link ;
procedure listinitialize;
   begin
      new(head) ; new(z) ;
      head ↑.next := z ; z ↑.next := z
   end ;
procedure deletenext(t:link);
   begin
      t ↑.next := t ↑.next ↑.next;
   end ;
procedure insertafter(v:integer ; t:link);
var x:link ;
   begin
      new(z) ;
      x ↑.key := v ; x ↑.next := t ↑.next ;
      t ↑.next := x
   end;
```

Dạng chính xác của các xâu được mô tả trong khai báo type: xâu được tạo bởi các nút, mỗi nút chứa một số nguyên và một liên kết (link) tới nút kế trên xâu. "Key" ở đây là một số nguyên để cho đơn giản, và có thể phức tạp tùy ý - mỗi liên kết là chìa khóa của xâu. Biến head là một liên kết tới nút đầu trên một xâu: ta có thể xem xét các nút còn lại theo thứ tự bằng cách đi theo các liên kết cho tới khi tìm tới z, liên kết mà nó chỉ tới nút giả biểu diễn cuối xâu. Cú pháp Pascal dùng cho việc dò theo các liên kết là "mũi tên lên" : ta viết một tham chiếu tới một liên kết, theo sau là ký hiệu này để chỉ ra một tham chiếu tới nút được trỏ bởi liên kết đó. Ví dụ, tham chiếu head↑.next↑.key tham khảo phần tử đầu tiên trên một xâu, và head↑.next↑.next↑.key tham khảo tới cái thứ hai.

Khai báo type đơn giản mô tả dạng của các nút ; các nút có thể được khởi tạo chỉ khi thủ tục có sẵn new được gọi. Ví dụ, lệnh gọi new (head) tạo ra một nút mới, đặt một con trỏ tới nó vào trong head. Mục đích của thủ tục này là để làm nhẹ bớt cho người lập trình gánh nặng phải "cấp phát" bộ nhớ cho nút khi xâu phình ra (ta sẽ bàn đến cơ chế này chi tiết hơn dưới đây). Có một thủ tục tương ứng có sẵn là dispose để xóa nút, nó có thể được dùng bởi thủ tục gọi, hay có lẽ nút, mặc dù đã được xóa khỏi một xâu, lại có thể được thêm vào một xâu khác.

Độc giả được khuyến khích để kiểm chứng các cài đặt Pascal này dựa trên các mô tả bằng ngôn ngữ tự nhiên đã được cho ở trên. Cụ thể, thật đáng nói ở chặng này là việc xem xét tại sao các nút giả lại là hữu ích. Đầu tiên, nếu quy ước là "head" trỏ tới đầu xâu thay vì có một nút head, thì thủ tục chèn (insert) sẽ cần một phép kiểm tra đặc biệt lúc chèn vào đầu xâu. Thứ hai là quy ước về z bảo vệ được thủ tục xóa (delete) tránh khỏi (ví dụ) một lệnh gọi để xóa một phần tử từ một xâu rỗng.

Một quy ước phổ biến khác cho việc chấm dứt một xâu là làm cho nút cuối trỏ tới nút đầu tiên, thay vì dùng các nút giả head và z. Xâu này được gọi là xâu vòng : nó cho phép một chương trình đi vòng trên xâu miễn là có một cái gì đó trong xâu đó. Dùng một nút giả để đánh dấu đầu (và cuối) xâu và để kiểm soát được trường hợp xâu rỗng thì đôi khi tiện lợi.

Có thể hỗ trợ thao tác "tìm phần tử nằm trước phần tử đã cho" bằng cách dùng một xâu liên kết kép, trong đó chúng ta duy trì hai mối liên kết cho mỗi một nút, một cái tới phần tử nằm trước, và một cái tới phần tử nằm sau. Cái giá của việc cung cấp khả năng phụ trợ này là tăng gấp đôi số thao tác mối liên kết cho mỗi một thao tác cơ bản, vì vậy thường nó không được dùng trừ phi đặc biệt được cần đến. Tuy nhiên, như đã lưu ý ở trên, nếu một nút bị xóa và chỉ có một liên kết tới nó là dùng được (có lẽ nó cũng là một phần tử của một CTDL khác nào đó), thì liên kết kép có thể được cần đến.

Ta sẽ thấy nhiều ví dụ về các ứng dụng của những thao tác này và các thao tác cơ bản khác trên các xâu liên kết trong các chương

sau. Vì các thao tác chỉ gồm một vài lệnh, thông thường ta thao tác các xâu một cách trực tiếp thay vì dùng các thủ tục chính xác ở trên. Như một ví dụ, ta sẽ xét tiếp một chương trình để giải "bài toán Josephus" theo tinh thần của sàng Eratosthenes.

Đối với bài toán Josephus, ta tưởng tượng rằng N người đã quyết định một cuộc tự sát tập thể bằng cách tự đứng trong một vòng tròn và giết người thứ M quanh vòng tròn, thu hẹp hàng ngũ lại khi từng người lần lượt ngã khỏi vòng tròn. Vấn đề là tìm xem người nào là người sẽ chết cuối cùng (mặc dù có thể người đó sẽ có một sự thay đổi quyết định vào phút chót !), hay tổng quát hơn là tìm ra thứ tự mà từng người sẽ bị giết. Ví dụ, nếu N = 9 và M = 5, thì những người bị giết sẽ là theo thứ tự 5 1 7 4 3 6 9 2 8. Chương trình sau đây đọc vào N và M rồi in ra thứ tự này :

```
program josephus (input, output);
type link = ↑node;
   node =   record key : integer; next : link end;
var i, N, M : integer; t, x : link;
begin
   read (N, M);
   new (t); t↑.key := 1; x := t;
   for i := 2 to N do
      begin new(t↑.next); t := t↑.next; t↑.key := i
      end;
   t↑.next := x;
   while t <> t↑.next do
   begin
      for i:= 1 to M-1 do t := t↑.next;
      write (t↑.next↑.key);
      x := t↑.next; t↑.next := t↑.next↑.next
       dispose (x);
   end;
   writeln(t↑.key);
end.
```

Chương trình dùng một xâu liên kết vòng để mô phỏng dãy các hành hình một cách trực tiếp. Đầu tiên, xâu được thiết lập với các khóa từ 1 đến N : biến x giữ vị trí bắt đầu của xâu khi nó được tạo ra, rồi con trỏ trong nút cuối cùng trong xâu được đặt về x. Sau đó, chương trình duyệt qua xâu, đếm qua M-1 phần tử rồi xóa cái kế

tiếp, cho đến khi chỉ còn lại một cái (chỉ tới chính nó). Lưu ý lệnh gọi dispose để xóa, tương ứng với một sự hành hình : lệnh này ngược với lệnh new đã đề cập ở trên.

VIỆC CẤP PHÁT BỘ NHỚ

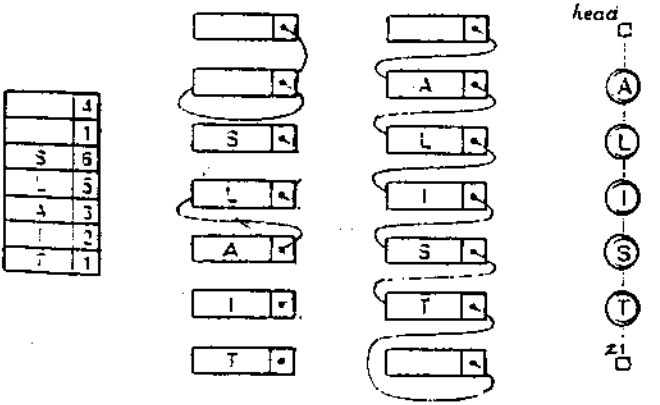
Các con trỏ của Pascal cung cấp một cách thuận tiện để cài đặt các xâu, như đã minh họa ở trên, nhưng có những cách khác nữa. Trong phần này ta sẽ thảo luận về việc làm thế nào dùng mảng để cài đặt các xâu liên kết và điều này có liên hệ như thế nào với biểu diễn thực sự của các xâu liên trong một chương trình Pascal. Như đã đề cập ở trên, mảng là một biểu diễn khá trực tiếp của bộ nhớ máy tính, như thế việc phân tích sự cài đặt ra sao một cấu trúc dữ liệu như một mảng sẽ cung cấp một vài hiểu biết sâu sắc về việc nó được biểu diễn trong máy tính ở cấp thấp như thế nào. Cụ thể, ta sẽ chú ý xem làm thế nào nhiều xâu có thể được biểu diễn một cách đồng thời.

Trong một biểu diễn trực tiếp bằng mảng của các xâu liên kết, ta sẽ dùng các chỉ mục thay cho các mối liên kết. Một cách để tiến hành là định nghĩa một mảng các mẩu tin (record) giống các mẩu tin ở trên nhưng dùng các số nguyên (cho các chỉ mục mảng) thay vì là các liên kết cho trường (field) kế tiếp. Một cách khác, thường thuận tiện hơn, là dùng "các mảng đồng hành" : ta lưu các phần tử trong một mảng key [1..N] và các liên kết trong một mảng next [1..N]. Vì vậy key[next [head]] liên hệ đến phần tử đầu tiên của danh sách, key [next [next [head]]] là phần tử thứ hai, ... Ưu điểm của việc dùng các mảng đồng hành là cấu trúc có thể được tạo dựng "trên đỉnh của" dữ liệu : mảng key chứa dữ liệu và chỉ chứa dữ liệu thôi - tất cả cấu trúc là trong mảng đồng hành next. Ví dụ, một xâu khác có thể được xây dựng bằng cách dùng cùng một mảng dữ liệu và một mảng "liên kết" đồng hành khác, hay các dữ liệu khác có thể thêm vào với các mảng đồng hành khác. Đoạn chương trình sau đây cài đặt các thao tác xâu cơ bản bằng cách dùng mảng đồng hành.

```
var key, next : array [0..N] of integer ; x, head, z : integer ;
procedure listinitialize ;
   begin
      head := 0 ; z := 1 ; x := 1 ;
      next [head] := z ; next [z] := z ;
   end ;
procedure deletenext (t : integer) ;
   begin
      next [t] := next [next[t]]
   end ;
procedure insertafter (v : integer ; t : integer);
   begin
      x := x+1 ;
      key [x] := v ; next [x] := next [t] ;
      next [t] := x
   end ;
```

"Con trỏ" x thay cho hàm cấp phát bộ nhớ new : nó lưu vết của vị trí kế chưa được dùng trong mảng.

Hình 3.5 minh họa làm thế nào xâu mẫu của chúng ta có thể được cài đặt bằng các mảng đồng hành, và biểu diễn này có liên hệ như thế nào với biểu diễn bằng hình vẽ mà ta đang dùng. Các



Hình 3.5 *Cài đặt mảng của một xâu liên kết*

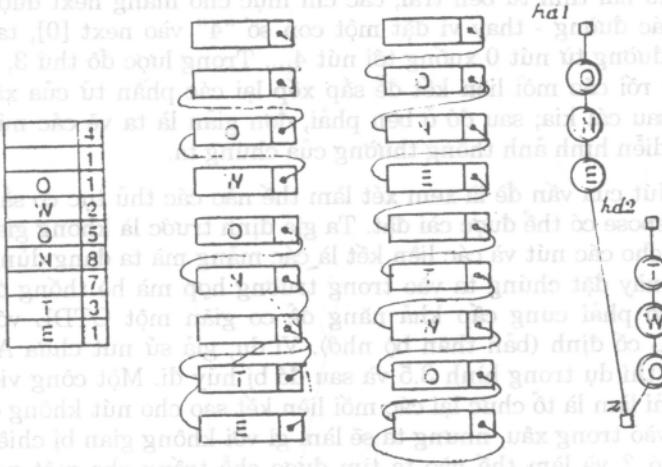
mảng key và next được chỉ ra ở bên trái, như chúng xuất hiện (lấy ví dụ) nếu S L A I T được chèn vào trong một xâu trống khởi đầu, với S, L, và A được chèn vào sau head ; I sau L, và T sau S. Vị trí 0 là head và vị trí 1 là z (những vị trí này được đặt bởi listinitialize) - vì next [0] là 4, nên phần tử đầu tiên trên xâu là key [4] (A) ; vì next [4] là 3, phần tử thứ hai trên xâu là key [3] (L),... Trong lược đồ thứ hai tính từ bên trái, các chỉ mục cho mảng next được thay bởi các đường - thay vì đặt một con số "4" vào next [0], ta sẽ vẽ một đường từ nút 0 xuống tới nút 4,... Trong lược đồ thứ 3, chúng ta gỡ rối các mối liên kết để sắp xếp lại các phần tử của xâu, cái này sau cái kia; sau đó ở bên phải, đơn giản là ta vẽ các nút theo biểu diễn hình ảnh thông thường của chúng ta.

Nút của vấn đề là xem xét làm thế nào các thủ tục có sẵn new và dispose có thể được cài đặt. Ta giả định trước là không gian duy nhất cho các nút và các liên kết là các mảng mà ta đang dùng : giả định này đặt chúng ta vào trong trường hợp mà hệ thống đang ở khi nó phải cung cấp khả năng để co giãn một CTDL với một CTDL cố định (bản thân bộ nhớ). Ví dụ, giả sử nút chứa A sẽ bị xóa ở thí dụ trong hình 3.5 và sau đó bị hủy đi. Một công việc mà ta phải làm là tổ chức lại các mối liên kết sao cho nút không còn bị móc vào trong xâu, nhưng ta sẽ làm gì với không gian bị chiếm bởi nút đó ? và làm thế nào ta tìm được chỗ trống cho một nút khi new được gọi và cần một chỗ trống khác ?

Sau khi suy nghĩ kỹ, độc giả sẽ thấy rằng lời giải là rõ ràng: một xâu liên kết sẽ được dùng để lưu giữ dấu vết của không gian còn trống ! ta coi xâu này như là "xâu trống". Sau đó, khi ta xóa một nút khỏi xâu của ta, ta sẽ hủy nó bằng cách chèn nó vào trong xâu trống, và khi ta cần một nút mới, ta sẽ nhận được nó bằng cách xóa nó khỏi xâu trống. Cơ chế này cho phép nhiều xâu khác nhau có thể chiếm cùng một mảng.

Một ví dụ đơn giản với hai xâu (nhưng không có xâu trống) được minh họa trong hình 3.6. Có hai nút đầu xâu hd1=0 và hd2=6, nhưng cả hai xâu có thể chia nhau cùng z (để xây dựng nhiều xâu, thủ tục listinitialize ở trên phải được sửa đổi để quản lý nhiều hơn một head). Bây giờ next [0] là 4, như thế phần tử đầu tiên trên xâu thứ nhất là key [4] (O) ; vì next [6] là 7, phần tử đầu tiên trên xâu thứ hai là key [7] (T),... Các hình khác trong hình 3.6

minh họa kết quả của việc thay thế các giá trị next bởi các đường, gỡ rối các nút, và thay đổi thành biểu diễn hình ảnh đơn giản của chúng ta, giống như trong hình 3.5. Cùng một kỹ thuật này có thể được dùng để duy trì nhiều xâu trong cùng mảng, một xâu trong chúng sẽ là một xâu trống, như đã mô tả ở trên.



Hình 3.6 *Hai xâu chia nhau cùng một không gian*

Khi việc quản lý bộ nhớ được cung cấp bởi hệ thống, như trong Pascal, thì không có lý do gì gạt bỏ nó đi để dùng phương pháp này. Mô tả ở trên dự định để chỉ ra làm thế nào thực hiện được việc quản lý bộ nhớ bởi hệ thống (nếu hệ thống của độc giả không thực hiện quản lý bộ nhớ, thì mô tả ở trên sẽ cung cấp một điểm khởi đầu cho một bản cài đặt). Vấn đề thực sự mà hệ thống phải đương đầu là khá phức tạp, vì không phải tất cả các nút nhất thiết là có cùng kích thước. Cũng vậy, một vài hệ thống làm nhẹ bớt cho người sử dụng nhu cầu phải hủy một cách tường minh các nút bằng cách dùng các thuật toán "dọn rác" để loại bỏ bất kỳ một nút nào mà nó không được tham chiếu bởi bất kỳ một mối liên kết nào. Một số các thuật toán quản lý bộ nhớ thông minh hơn đã được phát triển để kiểm soát hai trường hợp này.

NGĂN XẾP ĐẨY XUỐNG (Pushdown stack)

Chúng ta đang tập trung vào việc cấu trúc các dữ liệu, nhằm mục đích để chèn, xóa, hay truy xuất các phần tử một cách tùy ý. Thực sự, hóa ra là đối với nhiều ứng dụng, đủ để xem xét các hạn chế khác nhau (khá nghiêm ngặt) trên việc làm thế nào truy xuất được các CTDL. Các hạn chế như vậy là có lợi theo hai cách : trước tiên, chúng có thể làm nhẹ bớt yêu cầu cho chương trình dùng CTDL là phải quan tâm đến các chi tiết của nó (ví dụ như việc giữ dấu vết của các liên kết hay các chỉ mục của các phần tử); thứ hai là chúng cho phép các cài đặt đơn giản và mềm dẻo hơn, vì cần ít thao tác hơn.

CTDL có truy xuất hạn chế quan trọng nhất là ngăn xếp đẩy xuống. Nó chỉ có hai thao tác cơ bản : người ta có thể cất (push) một phần tử lên ngăn xếp (chèn ở đầu) và lấy (pop) một phần tử (loại bỏ nó khỏi đầu ngăn xếp). Một ngăn xếp thao tác giống như một tủ đựng của một quản trị viên bận rộn : công việc được chất đống trong một ngăn xếp, và bất kỳ lúc nào quản trị viên sẵn sàng để làm một công việc nào đó, thì anh ta sẽ lấy nó ra khỏi ngăn xếp ở trên cùng. Điều này có thể hiểu là một việc gì đó đã bị xếp ở đáy ngăn xếp một khoảng thời gian nào đó, nhưng một quản trị viên tốt sẽ quản lý để làm sao cho ngăn xếp được trống thường xuyên. Hóa ra là một chương trình máy tính đôi khi được tổ chức một cách tự nhiên theo phương pháp này, trì hoãn lại một vài tác vụ nào đó trong khi đang thực hiện những cái khác, và vì vậy các ngăn xếp đẩy xuống xuất hiện như là một CTDL cơ sở cho nhiều thuật toán.

Ta sẽ thấy nhiều ứng dụng lớn của ngăn xếp trong các chương sau : đối với một ví dụ dẫn nhập, ta hãy xem việc dùng các ngăn xếp để lượng giá các biểu thức số học. Giả sử người ta muốn tìm giá trị của một biểu thức số học đơn giản gồm việc nhân và cộng các số nguyên, ví dụ như :

5 * (((9+8) * (4*6)) + 7)

Một ngăn xếp là cơ chế lý tưởng để cất các kết quả trung gian trong một phép tính như vậy. Ví dụ ở trên có thể được tính với các lệnh gọi :

```
push(5);
push(9);
push(8);
push(pop+pop);
push(4);
push(6);
push(pop*pop);
push(pop*pop);
push(7);
push(pop+pop);
push(pop*pop);
writeln(pop);
```

Thứ tự trong đó các phép toán thực hiện được chỉ bởi các dấu ngoặc đơn trong biểu thức, và bởi quy ước mà ta tiến hành từ trái sang phải. Các quy ước khác là có thể được; ví dụ 4*6 có thể được tính trước 9+8 trong ví dụ trên.

Một vài máy cộng số học và một vài ngôn ngữ tính dựa trên phương pháp tính toán trên các thao tác ngăn xếp của chúng một cách tường minh theo cách này : mỗi phép toán lấy các tham số của nó khỏi ngăn xếp và trả về các kết quả của nó cho ngăn xếp. Như ta sẽ thấy trong chương 5, các ngăn xếp thường xuất hiện ngầm ngay cả khi không được dùng một cách tường minh.

Các thao tác ngăn xếp cơ bản thì dễ cài đặt bằng cách dùng xâu liên kết, như trong cài đặt ở trang sau.

Cài đặt này cũng gồm đoạn chương trình để khởi động một ngăn xếp và để kiểm tra xem nó có rỗng hay không). Trong một ứng dụng mà chỉ có một ngăn xếp được sử dụng, thì chúng ta có thể giả định là biến toàn cục head là liên kết tới ngăn xếp ; nếu không, các cài đặt có thể được sửa đổi để cũng đưa một liên kết tới ngăn xếp.

Thứ tự tính toán trong ví dụ số học ở trên yêu cầu các toán hạng xuất hiện trước toán tử sao cho chúng có thể ở trên ngăn xếp khi toán tử bị bắt gặp. Bất kỳ một biểu thức số học nào cũng có thể được viết lại theo cách này - ví dụ ở trên tương ứng với biểu thức

5 9 8 + 4 6 * * 7 + *

```
type link = ↑node;
   node =  record key : integer; next : link
           end;
var head, z : link;
procedure stackinit;
   begin
      new(head); new(z);
      head↑.next := z; z↑.next := z
   end;
procedure push (v : integer);
var t : link;
   begin
      new(t);
      t↑.key := v; t↑.next := head↑.next;
      head↑.next := t
   end;
procedure pop : integer;
var t : link;
   begin
      t := head↑.next;
      pop := t↑.next;
      head↑.next := t↑.next;
      dispose(t)
   end;
function stackempty : boolean;
   begin stackempty := (head↑.next = z)
   end;
```

Cách này được gọi là ký hiệu Balan ngược (vì nó được giới thiệu bởi một nhà luận lý học người Balan), hay postfix. Cách viết thông thường của các biểu thức số học được gọi là infix. Một tính chất thú vị của postfix là không cần đến các dấu ngoặc đơn ; trong infix, chúng được cần đến để phân biệt, ví dụ như 5*(((9+8)*(4*6))+7) với ((5*9)+8) * ((4*6)+7). Chương trình sau đây chuyển một biểu thức infix hợp lệ có ngoặc đầy đủ thành một biểu thức postfix :

```
stackinit ;
repeat
   repeat read(c) until c<>' ';
   if c=')' then write(chr(pop)) ;
   if c='+' then push(ord(c));
   if c='*' then push(ord(c));
   while (c>='0') and (c<='9') do
      begin write(c) ; read(c) end ;
   if c<>'(' then write (' ') ;
until eoln ;
```

Các đối số đơn giản được cho qua, vì chúng xuất hiện trong biểu thức postfix theo cùng thứ tự như trong biểu thức infix. Sau đó mỗi dấu ngoặc phải chỉ ra rằng cả hai đối số cho toán tử cuối đã được xuất ra, vì vậy chính toán tử đó có thể được lấy ra và ghi ra. (chương trình này không kiểm các lỗi trong lúc nhập và cần các khoảng trống giữa các toán tử, các dấu ngoặc đơn và các toán hạng).

Lý do chính để dùng postfix là việc lượng giá có thể được thực hiện theo một phương thức rất đơn giản với một ngăn xếp, như trong chương trình sau :

```
stackinit ;
repeat
   x := 0 ;
   repeat read(c) until c <> ' ';
   if c='*' then x := pop * pop ;
   if c='+' then x := pop + pop ;
   while (c>='0') and (c<='9') do
      begin x := 10 * x + (ord(c) - ord('0')) ; read(c)
      end ;
   push(x) ;
until eoln ;
writeln(pop) ;
```

Chương trình này đọc bất kỳ một biểu thức nào gồm có phép nhân và phép cộng số nguyên, sau đó in giá trị của biểu thức. Các khoảng trống được bỏ qua, và vòng lặp while chuyển các số nguyên từ dạng ký tự thành số để tính. Ngoài ra, thao tác của chương trình là đơn giản. Các số nguyên (toán hạng) được cất trên ngăn

xếp và phép nhân và cộng thay thế hai phần tử trên đỉnh ngăn xếp bởi kết quả của phép toán.

Nếu kích thước tối đa của một ngăn xếp có thể dự đoán trước được, thì nó có thể thích hợp để dùng một biểu diễn mảng thay vì dùng một xâu liên kết, như trong cài đặt sau :

```
const maxP = 100 ;
var   stack : array [0..maxP] of integer ;
      p : integer ;
procedure push (v : integer) ;
   begin stack[p] := v ; p := p+1 end ;
function pop : integer ;
   begin p := p-1 ; pop := stack[p] end ;
procedure stackinit ;
   begin p := 0 end ;
function stackempty : boolean ;
   begin stackempty := (p<=0) end ;
```

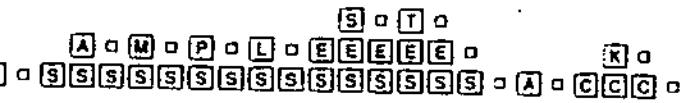
Biến p là một biến toàn cục mà nó duy trì vết của vị trí đỉnh của ngăn xếp. Đây là một cài đặt rất đơn giản mà nó tránh được việc dùng không gian phụ trợ cho các mối liên kết, ở cái giá có lẽ là phí chỗ do phải dành riêng không gian trống cho ngăn xếp kích thước lớn nhất.

Hình 3.7 minh họa làm thế nào một ngăn xếp mẫu phát triển qua một chuỗi các thao tác push và pop được biểu diễn bởi dãy :

A * S A * M * P * L * E S * T * * * A * C K * *

Sự xuất hiện của một chữ cái trong danh sách này có nghĩa là "push" (chữ cái) ; dấu hoa thị có nghĩa là "pop".

Đặc biệt, một số lượng lớn các thao tác sẽ chỉ cần đến một ngăn xếp nhỏ. Nếu ta tin tưởng vào điều này, thì một biểu diễn mảng sẽ được cần đến. Nếu không, thì một xâu liên kết có thể cho phép ngăn xếp co giãn một cách nhịp nhàng, đặc biệt nếu nó là một trong nhiều CTDL như vậy.



Hình 3.7 *Các đặc trưng động của một ngăn xếp*

HÀNG ĐỢI (Queue)

Một CTDL cơ bản khác có truy xuất hạn chế được gọi là hàng đợi. Cũng vậy, nó chỉ gồm có hai thao tác cơ bản: người ta có thể chèn (insert) một phần tử vào trong hàng đợi ở đầu và lấy đi (remove) một phần tử ở cuối. Có lẽ tủ đựng của quản trị viên bận rộn của chúng ta nên thao tác giống như một hàng đợi, vì như vậy công việc mà nó đến đầu tiên sẽ được thực hiện trước. Trong một ngăn xếp, một việc gì đó có thể bị chôn vùi ở đáy ngăn xếp, nhưng trong một hàng đợi mọi thứ được xử lý theo thứ tự nhận được.

Mặc dù ngăn xếp được gặp nhiều hơn là hàng đợi do mối quan hệ cơ bản của nó với sự đệ quy (xem chương 5), chúng ta sẽ gặp các thuật toán mà đối với chúng hàng đợi là một CTDL tự nhiên. Các ngăn xếp đôi khi được xem như tuân theo một quy luật là "vào sau, ra trước" (LIFO last in, first out) ; hàng đợi tuân theo quy luật "vào trước, ra trước (FIFO : first in, first out).

Cài đặt theo xâu liên kết của các thao tác hàng đợi thì đơn giản và được để lại như một bài tập cho người đọc. Như với ngăn xếp, một mảng cũng có thể được dùng nếu người ta có thể lượng định được kích thước lớn nhất, như trong cài đặt sau đây :

```
const max = 100 ;
var queue : array [0..max] of integer ; head, tail : integer ;
procedure put (v : integer) ;
   begin
      queue[tail] := v ; tail := tail+1 ;
      if tail>=max then tail := 0
   end ;
function get : integer ;
begin
   get := queue[head] ; head := head+1 ;
   if head>max then head := 0
end ;
procedure queueinitialize ;
   begin head := 0 ; tail := 0 end;
function queueempty : boolean ;
   begin queueempty := (head = tail) end ;
```

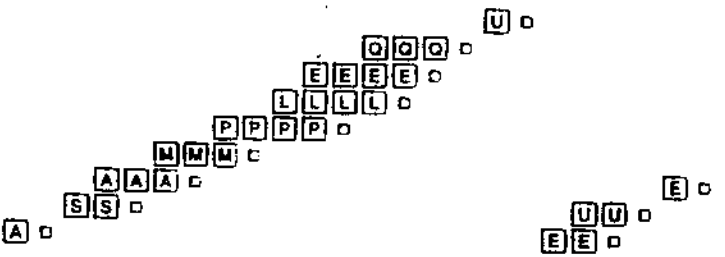
Cần phải duy trì hai chỉ mục, một tới nơi bắt đầu của hàng đợi (head) và một tới cuối (tail). Nội dung của hàng đợi là tất cả các phần tử trong mảng nằm giữa head và tail, có thể "quay vòng" trở lại 0 khi đến cuối mảng. Nếu head = tail thì hàng đợi được xem như là rỗng ; nếu head = tail+1, hay tail = max và head = 0 thì nó được xem như là đầy.

Hình 3.8 minh họa làm thế nào một hàng đợi mẫu phát triển qua một chuỗi các thao tác get và put được biểu diễn bởi dãy :

A * S A * M * P * L E * Q * * * U * E U * * E *

Sự xuất hiện của một chữ cái trong danh sách này có nghĩa là "put" (chữ cái) ; dấu hoa thị có nghĩa là "get".

Trong chương 20 chúng ta gặp một "deque" (hay "hàng đợi hai đầu": double ended queue), mà nó là một tổ hợp của một ngăn xếp và một hàng đợi, và trong các chương 4 và 30 ta sẽ bàn về các ví dụ khá cơ bản ám chỉ việc áp dụng một hàng đợi như là một cơ chế để cho phép khảo sát cây và đồ thị.



Hình 3.8 *Các đặc tính động của một hàng đợi*

KIỂU DỮ LIỆU TRỪU TƯỢNG

Chúng ta đã thấy ở trên, thường thuận tiện khi mô tả các thuật toán và CTDL theo các thao tác được thực hiện, thay vì theo các chi tiết của việc cài đặt. Khi một CTDL được định nghĩa theo cách này, nó được gọi là một kiểu dữ liệu trừu tượng. Ý tưởng là để tách bạch "quan niệm" về những gì mà CTDL nên làm ra khỏi bất kỳ một cài đặt cụ thể nào.

Đặc trưng xác định của một kiểu dữ liệu trừu tượng là không có gì nằm bên ngoài các định nghĩa của CTDL và các thuật toán thao tác trên nó là có thể tham khảo được tới bất kỳ một thứ gì nằm bên trong, ngoại trừ là qua các lệnh gọi hàm và thủ tục cho các thao tác cơ bản. Động cơ chính cho việc phát triển các kiểu dữ liệu trừu tượng đã như là một cơ chế dùng cho việc tổ chức các chương trình lớn. Chúng cung cấp một phương pháp để giới hạn kích thước và độ phức tạp của giao tiếp giữa các thuật toán (có thể là phức tạp) cùng các CTDL tương ứng với các chương trình (có thể là một số lượng lớn các chương trình) mà nó dùng các thuật toán và CTDL. Điều này khiến cho việc hiểu chương trình lớn một cách dễ dàng hơn, và làm cho việc thay đổi và nâng cấp các thuật toán cơ bản trở nên thuận tiện hơn.

Các ngăn xếp và hàng đợi là các ví dụ cổ điển về các kiểu dữ liệu trừu tượng : hầu hết các chương trình chỉ cần được quan tâm về một vài thao tác cơ bản được định nghĩa tốt, chứ không phải là các chi tiết về các mối liên kết và các chỉ mục.

Mảng và xâu liên kết đến lượt nó có thể được xem như là sự tinh chế của một kiểu dữ liệu trừu tượng cơ bản được gọi là xâu tuyến tính. Mỗi cái trong chúng có thể hỗ trợ các thao tác như chèn, xóa, và truy xuất trên một cấu trúc cơ bản bên dưới gồm các phần tử có thứ tự tuần tự. Các thao tác này đủ để mô tả các thuật toán, và sự trừu tượng hóa xâu tuyến tính có thể là hữu ích trong những chặng đầu của sự phát triển thuật toán. Nhưng như chúng ta đã thấy, mối quan tâm của người lập trình là định nghĩa một cách cẩn thận các thao tác nào sẽ được dùng, đối với các cài đặt khác nhau, có thể có các đặc trưng về hiệu năng hoàn toàn khác nhau. Ví dụ, việc dùng một xâu liên kết thay cho một mảng đối với sàng Eratosthenes sẽ là tốn kém vì tính hiệu quả của thuật toán phụ thuộc vào khả năng đi từ bất kỳ một vị trí nào của mảng tới bất kỳ một vị trí khác của mảng một cách nhanh chóng, và việc dùng một mảng thay vì một xâu liên kết cho bài toán Josephus sẽ là tốn kém vì tính hiệu quả của thuật toán phụ thuộc vào sự biến mất của các phần tử bị xóa.

Nhiều thao tác khác tự chúng nẩy sinh ra trên các xâu tuyến tính mà nó cần đến nhiều thuật toán và các CTDL tinh vi hơn để

hỗ trợ chúng một cách hiệu quả. Hai thao tác quan trọng nhất là sắp các phần tử theo thứ tự tăng của khóa của chúng (chủ đề của các chương 8 đến 13) và thao tác tìm một phần tử với một khóa cụ thể (chủ đề của các chương 14 đến 18).

Một kiểu dữ liệu trừu tượng có thể được dùng để định nghĩa một kiểu dữ liệu trừu tượng khác : ta dùng xâu liên kết và mảng để định nghĩa ngăn xếp và hàng đợi. Thực vậy, chúng ta dùng các trừu tượng "con trỏ" và "mẩu tin" được cung cấp bởi Pascal để xây dựng các xâu liên kết, và dùng kiểu trừu tượng "mảng" được cung cấp bởi Pascal để tạo nên các mảng. Thêm vào đó, ta đã thấy ở trên là ta có thể xây dựng các xâu liên kết bằng mảng, và ta sẽ thấy trong chương 36 là mảng sẽ đôi khi được xây dựng bằng xâu liên kết ! Sức mạnh thực sự của khái niệm kiểu dữ liệu trừu tượng là nó cho phép ta dễ dàng cấu trúc nên các hệ thống lớn trên các mức độ trừu tượng khác nhau, từ các lệnh ngôn ngữ máy được cung cấp bởi máy tính, đến các khả năng khác nhau được cung cấp bởi ngôn ngữ lập trình, để sắp xếp, tìm kiếm và các khả năng cấp cao hơn khác được cung cấp bởi các thuật toán được bàn đến trong quyển sách này, thậm chí tới các cấp trừu tượng cao hơn nữa mà ứng dụng có thể đề nghị.

Trong cuốn sách này, chúng ta đối diện với các chương trình tương đối nhỏ được tích hợp khá chặt với các cấu trúc dữ liệu tương ứng của chúng. Trong khi có thể nói về sự trừu tượng hóa ở giao diện giữa các thuật toán của ta với các cấu trúc dữ liệu của thuật toán đó, thực ra sẽ thích hợp hơn khi tập trung vào các cấp trừu tượng hóa cao hơn (gần gũi hơn với ứng dụng) : khái niệm về sự trừu tượng hóa không nên làm chúng ta sao lãng việc tìm lời giải hiệu quả nhất cho một bài toán cụ thể. Ở đây ta coi quan điểm về hiệu năng mới là vấn đề chính! Các chương trình được phát triển với cách nhìn này trong đâu sau này có thể được dùng với một sự tin tưởng trong việc phát triển các cấp trừu tượng hóa cao hơn cho các hệ thống lớn.

Cho dù các kiểu dữ liệu trừu tượng có được sử dụng một cách tường minh hay không, chúng ta không thoát khỏi bổn phận phải phát biểu một cách chính xác những gì thuật toán của chúng ta làm. Thực ra, thường là thuận lợi để định nghĩa các giao diện với

các thuật toán và cấu trúc dữ liệu được cung cấp ở đây như là các kiểu dữ liệu trừu tượng; các ví dụ về điều này được tìm thấy trong các chương 11 và 14. Hơn nữa, người sử dụng của các thuật toán và cấu trúc dữ liệu có bổn phận phát biểu rõ ràng những gì anh ta muốn chúng phải làm - việc truyền thông một cách thích hợp giữa người sử dụng của thuật toán và người cài đặt thuật toán đó (ngay cả nếu họ là cùng một người) sẽ là chìa khóa để thành công trong việc xây dựng các hệ thống lớn. Các môi trường lập trình mà nó hỗ trợ việc phát triển các hệ thống lớn có những phương tiện mà nó cho phép điều này sẽ được thực hiện theo một phương pháp có hệ thống.

Như đã đề cập ở trên, các cấu trúc dữ liệu thực hiếm khi chỉ gồm có các số nguyên và các mối liên kết. Các nút thông thường chứa một lượng lớn các thông tin và có thể thuộc về nhiều cấu trúc dữ liệu độc lập. Ví dụ, một hồ sơ về dữ liệu nhân sự có thể chứa các mẩu tin với tên, địa chỉ, và các mẩu thông tin khác về các nhân viên, và mỗi mẩu tin có thể cần nằm trong một cấu trúc dữ liệu dùng cho việc tìm kiếm các nhân viên cụ thể, một CTDL khác cho việc trả lời các câu hỏi thống kê, ... Thậm chí có thể tạo dựng nên các cấu trúc dữ liệu thật phức tạp chỉ cần dùng đến các cấu trúc dữ liệu đơn giản đã được mô tả trong chương này : các mẩu tin ghi có thể lớn hơn và phức tạp hơn, nhưng các thuật toán là như nhau. Cũng vậy, chúng ta cần chú ý là chúng ta không phát triển các thuật toán chỉ tốt cho các mẩu tin nhỏ: ta sẽ trở lại vấn đề này ở cuối chương 8 và ở đầu chương 14.

BÀI TẬP

1. Viết một chương trình để điền vào một mảng 2 chiều bằng cách đặt a[i,j] là true nếu USCLN của i và j là 1 và false nếu ngược lại.

2. Cài đặt một thủ tục movenexttofront(t:link) cho một xâu liên kết mà nó chuyển nút nằm sau nút được trỏ tới bởi t đến nơi bắt đầu của xâu (hình 3.3 là một ví dụ của thao tác này cho những trường hợp đặc biệt khi t chỉ tới nút kế chót trong xâu).

3. Cài đặt một thủ tục exchange(t,u:link) cho một xâu liên kết mà nó hoán vị các nút được trỏ tới bởi t và u.

4. Viết một chương trình để giải bài toán Josephus, dùng một mảng thay vì dùng xâu liên kết.

5. Viết các thủ tục để chèn và xóa trong một xâu liên kết kép.

6. Viết các thủ tục cho một cài đặt bằng xâu liên kết của một ngăn xếp đẩy xuống, nhưng dùng các mảng đồng hành.

7. Hãy cho biết nội dung của ngăn xếp sau mỗi thao tác trong dãy E A S * Y * * Q U E * * * S T * * * I * O N * *. Ở đây một chữ cái có nghĩa là "push" (chữ cái) và "*" nghĩa là "pop".

8. Hãy cho biết nội dung của hàng đợi sau mỗi thao tác trong dãy E A S * Y * * Q U E * * * S T * * * I * O N * *. Ở đây một chữ cái hiểu là "put" (chữ cái) và "*" nghĩa là "get".

9. Hãy cho một dãy các lệnh gọi tới listinitialize, deletenext, và insertafter mà nó có thể sinh ra hình 3.5.

10. Dùng một xâu liên kết, cài đặt các thao tác cơ bản cho một hàng đợi.

4
CÂY

Các cấu trúc đã thảo luận trong chương 3 là cấu trúc một chiều: mỗi phần tử đi theo sau phần tử khác. Trong chương này chúng ta sẽ khảo sát các cấu trúc liên kết hai chiều được gọi là cây, đây là cấu trúc trung tâm của hầu hết các cấu trúc và các thuật toán quan trọng nhất của chúng ta. Thảo luận đầy đủ về cây có lẽ cần cả một cuốn sách đầy, chúng đã ra đời và phát triển mạnh trong nhiều ứng dụng bên ngoài khoa học máy tính và đã được nghiên cứu rộng rãi như các đối tượng toán học. Quyển sách này sẽ thảo luận các phương pháp cơ bản về cây khi cần trong mỗi chương của nó. Trong chương này, chúng ta đưa ra các định nghĩa cơ sở và các từ ngữ liên quan đến cấu trúc cây, kiểm tra một vài tính chất quan trọng và nghiên cứu việc biểu diễn chúng trên trên máy tính. Trong các chương kế, chúng ta sẽ xem nhiều thuật toán thao tác trên những cấu trúc dữ liệu cơ bản này.

Cây được gặp thường xuyên trong cuộc sống của chúng ta, và chắc chắn độc giả quen thuộc với khái niệm cơ sở. Ví dụ, một số người theo dõi tổ tiên và con cháu với cây gia phả: như chúng ta sẽ thấy, phần nhiều các từ ngữ của chúng ta được thừa kế từ sự sử dụng này. Một ví dụ khác được thấy trong việc tổ chức các cuộc thi đấu thể thao, chúng ta sẽ gặp vấn đề này trong chương 11 đã được nghiên cứu bởi Lewis Carroll. Một ví dụ thứ ba được thấy trong sơ đồ tổ chức của một liên đoàn lớn, điều này gợi ý đến "sự phân cấp" được thấy trong nhiều ứng dụng của khoa học máy tính. Một ví dụ thứ tư là "cây phân tích" của một câu tiếng Anh bao gồm các phần cơ sở của câu, điều này liên hệ mật thiết đến xử lý các ngôn ngữ của máy tính, như sẽ thảo luận xa hơn trong chương 21. Các ví dụ khác sẽ được bàn đến xuyên qua quyển sách này.

THUẬT NGỮ

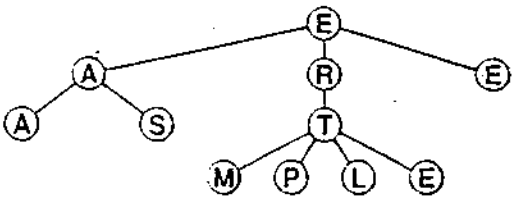
Chúng ta bắt đầu thảo luận về cây bằng cách định nghĩa chúng như các đối tượng trừu tượng và giới thiệu hầu hết các thuật ngữ cơ sở. Có nhiều cách định nghĩa cây tương đương với nhau và một số tính chất toán học mà hàm ý sự tương đương này; những điều này sẽ được thảo luận chi tiết hơn trong phần tới.

Một cây là một tập hợp khác trống gồm các đỉnh và các cạnh mà thỏa mãn các đòi hỏi nhất định. Một đỉnh là một đối tượng đơn (cũng được gọi là nút) có thể có một tên và có thể mang các thông tin khác; một cạnh thì thì nối hai đỉnh với nhau. Một đường đi trong một cây là một danh sách các đỉnh phân biệt mà các đỉnh liên tiếp nhau được nối bởi các cạnh trong cây. Một nút trong cây được chỉ định là gốc, tính chất định nghĩa của một cây là có chính xác một đường đi nối gốc và mỗi nút khác trong cây. Nếu có nhiều hơn một đường đi giữa gốc và một nút nào đó hay không có đường đi nào giữa gốc và một nút nào đó thì chúng ta sẽ có một đồ thị (xem chương 29) chứ không phải là một cây. Hình 4.1 cho thấy một ví dụ của cây.

Mặc dù định nghĩa hàm ý không có "hướng" đối với các cạnh, nhưng thông thường chúng ta nghĩ rằng các cạnh chỉ ra ngoài từ gốc (hướng xuống dưới như trong hình 4.1) hay hướng tới gốc (hướng lên như trong hình 4.1) tùy thuộc vào áp dụng cụ thể. Thông thường chúng ta vẽ cây với gốc ở đỉnh trên (mặc dù điều này có vẻ không tự nhiên), và chúng ta nói nút y ở phía dưới nút x (và x ở phía trên nút y) nếu x nằm trên con đường từ từ y tới gốc (nghĩa là nếu y ở dưới x như hình vẽ trên giấy và y được nối tới x bởi một con đường không đi qua gốc). Mỗi nút (ngoại trừ nút gốc) có chính xác một nút trên nó và được gọi là cha của nó, các nút ngay bên dưới trực tiếp một nút được gọi làm các con của nút đó. Đôi khi chúng ta tương tự hóa với cây gia phả và cũng đề cập đến "ông" hay "anh em" của một nút; trong hình 4.1 thì P là cháu của R và P có ba anh em.

Các nút không có con được gọi là các **nút lá** hay các **nút tận cùng**. Để đáp ứng với việc sử dụng sau này, các nút có ít nhất một con được gọi là các nút **không tận cùng**. Các nút tận cùng thì

thường khác với các nút không tận cùng, chẳng hạn chúng không có tên hay không có thông tin kết hợp với chúng. Đặc biệt trong các hoàn cảnh như thế, chúng ta xem các nút không tận cùng là các **nút trong** và các nút tận cùng là các **nút ngoài**.



Hình 4.1 *Một cây mẫu*

Bất kỳ nút nào đều cũng là gốc của một cây con bao gồm chính nút đó và các nút bên dưới nó. Cây trong hình 4.1 có bảy cây con một nút, một cây con ba nút, một cây con năm nút, và một cây con sáu nút. Một tập hợp các cây được gọi là một rừng; ví dụ nếu chúng ta xóa gốc và các cạnh nối tới gốc khỏi cây trong hình 4.1 thì chúng ta sẽ có một rừng bao gồm ba cây có gốc ở A, R, và E.

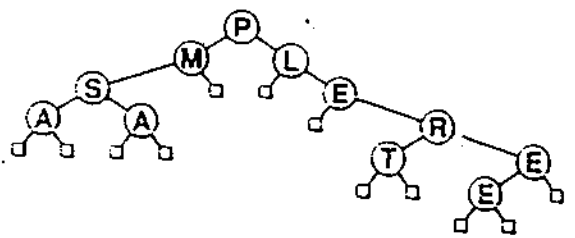
Đôi khi sự sắp xếp thứ tự của các con của mỗi nút thì quan trọng, và đôi khi điều này lại không quan trọng. Một **cây được sắp** là một cây mà thứ tự của các con của mỗi nút được quy định. Dĩ nhiên, các con được đặt trong một thứ tự nào đó khi ta vẽ một cây, và rõ ràng có nhiều phương pháp khác nhau để vẽ các cây không được sắp. Như chúng ta sẽ thấy bên dưới, điều này sẽ quan trọng khi chúng ta khảo sát việc biểu diễn các cây trong máy tính, bởi vì có rất ít tính linh động khi biểu diễn các cây được sắp. Hiển nhiên tùy với từng ứng dụng cụ thể mà người ta sẽ quy định chọn dạng cây nào cho phù hợp.

Các nút trong một cây được chia thành **các tầng**: tầng của một nút là số nút trên con đường từ nó tới gốc (không kể chính nút đó). Do đó, ví dụ như trong hình 4.1 thì R ở tầng 1 và S ở tầng 2. **Chiều cao** của một cây là tầng lớn nhất trong số tất cả các nút trong cây (hay là khoảng cách tối đa từ gốc tới một nút bất kỳ). Độ dài đường đi của một cây là tổng của tất cả các tầng của các nút

trong cây (hay là tổng các chiều dài của các con đường từ mỗi nút tới gốc). Cây trong hình 4.1 có chiều cao 3 và độ dài 21. Nếu tất cả các nút trong được phân biệt với tất cả các nút ngoài, chúng ta có **độ dài đường đi trong** và **độ dài đường đi ngoài**.

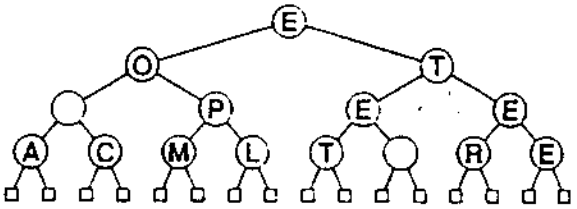
Nếu mỗi nút bị bắt buộc có một số cụ thể các con được xuất hiện theo một thứ tự cụ thể thì chúng ta có một **cây đa hướng.** Trong một cây như thế, người ta thường định nghĩa các nút ngoài đặc biệt mà không có con (và thông thường không có con cũng như không có các thông tin khác). Khi đó, các nút ngoài xem như các nút "nháp" (dummy) dành cho việc tham chiếu bởi các nút mà không có số lượng con cụ thể.

Dạng đơn giản nhất của cây đa hướng là **cây nhị phân**. Một cây nhị phân là một cây được sắp bao gồm hai dạng nút: các nút ngoài (không có con) và các nút trong (có chính xác hai con). Một ví dụ về cây nhị phân được cho thấy trong Hình 4.2. Bởi vì hai con của mỗi nút trong được sắp xếp thứ tự, chúng ta lưu ý đến con trái và con phải của các nút trong: mỗi nút trong phải có cả con trái lẫn con phải, mặc dù một trong hai con hay cả hai con có thể là một nút ngoài.



Hình 4.2 *Một cây nhị phân mẫu*

Mục đích của cây nhị phân là cấu trúc các nút trong, các nút ngoài chỉ đóng vai trò như những người giữ chỗ. Chúng ta đề cập đến chúng trong định nghĩa bởi vì các biểu diễn được dùng thông dụng nhất cho các cây nhị phân phải kể tới mỗi nút ngoài. Một cây nhị phân có thể "rỗng", nghĩa là không có một nút trong nào và chỉ có một nút ngoài.



Hình 4.3 *Một cây nhị phân đầy đủ*

Một **cây nhị phân đầy** là một cây mà các nút trong của nó trải hoàn toàn mỗi tầng, tầng cuối cùng có thể không trải hoàn toàn. Một **cây nhị phân đầy đủ** là một cây nhị phân đầy mà tất cả các nút trong trên tầng đáy xuất hiện ở bên trái của tất cả các nút ngoài trên tầng đó. Hình 4.3 cho thấy một ví dụ về cây nhị phân đầy đủ. Như chúng ta sẽ thấy, các cây nhị phân xuất hiện rộng rãi trong các ứng dụng máy tính, và tính năng của chúng tốt nhất khi chúng đầy (hay gần đầy). Trong chương 11 chúng ta sẽ kiểm tra một cấu trúc dữ liệu quan trọng được dựa trên các cây nhị phân đầy đủ.

Độc giả nên chú ý cẩn thận rằng, mặc dù mỗi cây nhị phân là một cây nhưng không phải cây nào cũng là cây nhị phân. Ngay cả khi xem xét các cây được sắp xếp mà mỗi nút có 0, 1 hay 2 con, mỗi cây như thế có thể tương ứng với nhiều cây nhị phân bởi vì các nút có một con có thể là trái hay phải trong một cây nhị phân.

Các cây thì liên hệ mật thiết với sự đệ qui mà chúng ta sẽ nghiên cứu trong trong chương kế tiếp. Thật ra, có lẽ phương pháp đơn giản nhất để định nghĩa các cây là phương pháp đệ qui như sau: "một cây thì hoặc là một nút đơn lẻ hay là một nút gốc được nối tới một tập hợp các cây" và "một cây nhị phân thì hoặc là một nút ngoài hay là một nút gốc (nút trong) được nối tới một cây nhị phân trái và một cây nhị phân phải"

CÁC TÍNH CHẤT

Trước khi khảo sát các biểu diễn của cây chúng ta tiếp tục về một số tính chất toán học quan trọng của cây. Có rất nhiều tính chất có thể nghiên cứu nhưng mục đích của chúng ta là quan tâm đến các tính chất liên quan đến các thuật toán trong quyển sách này.

TÍNH CHẤT 4.1 *Có đúng một con đường nối hai nút bất kỳ trong một cây.*

Hai nút bất kỳ của cây thì có ít nhất một tổ tiên chung gần nhất, đó chính là nút nằm trên đường đi từ cả hai nút tới gốc, hơn nữa không có nút con nào có cùng tính chất như vậy. Ví dụ: O là tổ trên chung gần nhất của C và L trong cây của hình 4.3. Tổ trên chung gần nhất phải luôn tồn tại bởi vì hoặc gốc chính là tổ trên chung gần nhất, hoặc cả hai nút nằm trong "cây con mới" có gốc nằm trong cây con nói trên. Có một đường đi từ mỗi nút tới tổ trên chung gần nhất, nối hai đường này ta được một đường đi nối hai nút.

Một hàm ý quan trọng của tính chất: là bất kỳ nút nào cũng có thể là gốc: mỗi nút trong cây có tính chất là có đúng một đường đi nối nút đó với mỗi nút khác trong cây. Trong định nghĩa của chúng ta thì gốc được xác định nên ta có một **cây xác định gốc** hay một **cây có hướng**, một cây mà gốc không xác định được gọi là một **cây tự do**.

TÍNH CHẤT 4.2 *Một cây N nút sẽ có N-1 cạnh*

Tính chất này suy ra từ sự quan sát rằng mỗi nút ngoại trừ nút gốc sẽ có một cha duy nhất và mỗi cạnh thì nối một nút nào đó tới cha nó. Chúng ta cũng có thể chứng minh tính chất này bằng phương pháp quy nạp nhờ vào định nghĩa đệ qui.

Hai tính chất kế tiếp mà chúng ta sẽ khảo sát thì liên quan đến các cây nhị phân. Như đã chú ý trước đây, cấu trúc này xuất hiện rất thường xuyên trong suốt quyển sách này, vì vậy hoàn toàn đáng giá khi chú ý đến những đặc trưng của chúng. Công việc này sẽ là nền tảng để hiểu các đặc trưng về tính năng của các thuật toán khác nhau mà chúng ta sẽ gặp.

TÍNH CHẤT 4.3 *Một cây nhị phân có N nút trong thì nó có N+1 nút ngoài.*

Tính chất này có thể được chứng minh bởi qui nạp. Một cây nhị phân không có nút trong thì có một nút ngoài, vì vậy tính chất đúng với N=0. Với N>0, bất kỳ cây nhị phân với N nút trong sẽ có những nút trong cây con trái của nó và N-1-k nút trong cây con phải của nó, trong đó k nằm giữa 0 và N-1, bởi vì gốc là một nút trong. Do giả thiết qui nạp, cây con trái có k+1 nút ngoài và cây con phải có N-k nút ngoài và ta có tổng cộng N+1 nút ngoài.

TÍNH CHẤT 4.4 *Chiều dài đường đi ngoài của một cây nhị phân N nút trong thì lớn hơn chiều dài đường đi trong 2N.*

Tính chất này cũng có thể được chứng minh bằng qui nạp, nhưng cũng có thể chứng minh cách khác. Chú ý rằng bất kỳ cây nhị phân nào cũng có thể được xây dựng theo xử lý sau đây: khởi đầu với cây nhị phân chỉ có một nút ngoài. Kế đến lập lại thao tác sau N lần: nhặt một nút ngoài nào đó và thay thế nó bởi một nút trong mới có hai con là nút ngoài. Nếu nút ngoài được chọn ở tầng k, chiều dài đường đi trong tăng lên k+1 (một nút ngoài ở tầng k bị xóa đi, nhưng hai nút ngoài ở tầng k+1 được thêm vào). Xử lý bắt đầu với một cây với chiều dài đường đi trong lẫn chiều dài đường đi trong lẫn chiều dài đường đi ngoài đều là 0, xử lý bao gồm N-1 bước và gia tăng độ dài đi ngoài nhiều hơn hai so với gia tăng độ đài đường đi trong ở mỗi bước.

Cuối cùng chúng ta khảo sát các tính chất đơn giản vào loại "bậc nhất" của cây nhị phân, đó là các cây nhị phân đầy. Những cây nhị phân loại này rất thú vị bởi vì chiều cao của chúng được bảo đảm thấp, vì vậy chúng ta không phải làm việc nhiều khi duyệt từ gốc tới một nút bất kỳ.

TÍNH CHẤT 4.5 *Chiều cao của cây nhị phân đầy N nút là khoảng $\log_2 N$.*

Tham khảo hình 4.3, nếu độ cao là n, thì chúng ta phải có

$2^n < N+1 <= 2^n$

bởi vì có N+1 nút ngoài. Điều này suy ra tính chất đã nêu

(Thông thường thì độ cao bằng đúng với số nguyên làm tròn gần $\log_2 N$ nhất, nhưng chúng ta muốn giữ chính xác như thảo luận trong chương 6).

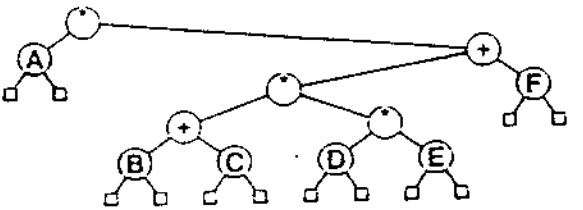
Các tính chất toán học sâu hơn về cây sẽ được thảo luận khi cần trong các chương sau. Bây giờ chúng ta chuẩn bị hướng tới vấn đề thực hành quan trọng là biểu diễn các cây trong máy tính và cách thao tác chúng thế nào để có hiệu quả.

BIỂU DIỄN CÁC CÂY NHỊ PHÂN

Biểu diễn thường dùng nhất cho các cây nhị phân là sử dụng các mẩu tin với hai liên kết cho mỗi nút. Thông thường chúng ta dùng liên kết được đặt tên là l và r (viết tắt cho "left" và "right") để chỉ rằng thứ tự được chọn cho biểu diễn thì tương ứng với cách vẽ cây trên trang giấy. Với một vài ứng dụng thì có thể thích hợp khi có hai dạng mẩu tin khác nhau, một dạng cho các nút trong và một dạng cho các nút ngoài; đối với các ứng dụng khác thì có thể thích hợp khi dùng cùng một dạng nút và dùng các liên kết của các nút ngoài cho các mục đích khác.

Để ví dụ cho việc sử dụng và xây dựng các cây nhị phân, chúng ta sẽ tiếp tục với ví dụ đơn giản trong chương trước về xử lý các biểu thức số học. Có một sự tương ứng cơ bản giữa các biểu thức số học và các cây như trong hình 4.4.

Chúng ta dùng các chỉ danh một ký tự thay vì những con số cho các đối số; lý lẽ cho vấn đề này sẽ được giải thích bên dưới. Cây phân tích cho một biểu thức được định nghĩa bởi quy luật đệ qui đơn giản: "Đặt toán tử ở gốc và kế đến đặt cây tương ứng với toán hạng đầu tiên của biểu thức ở bên trái và cây tương ứng với biểu thức của toán hạng thứ hai ở bên phải". Hình 4.4 là cây phân tích cho biểu thức ABC + DE * * F + * (dạng postfix), infix và postfix là hai phương pháp biểu diễn các biểu thức số học, các cây phân



Hình 4.4 *Cây phân tích cho biểu thức A*(((B+C)*(D*E))+F)*

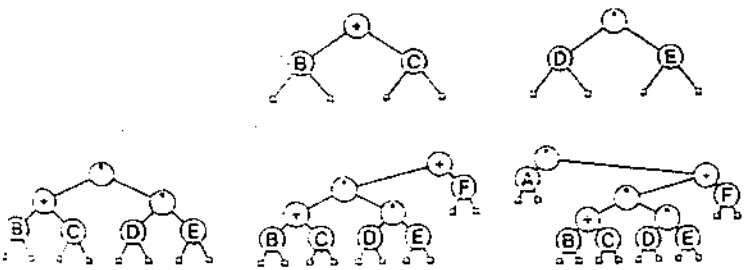
tích là phương pháp thứ ba.

Bởi vì mỗi toán tử tương ứng với chính xác hai toán hạng nên cây nhị phân thích hợp cho biểu thức loại này. Các biểu thức phức tạp hơn có lẽ đòi hỏi một dạng cây khác. Chúng ta sẽ trở lại vấn đề này chi tiết hơn trong chương 21; mục đích của chúng ta ở đây là xây dựng một biểu diễn cây của biểu thức số học.

Chương trình sau sẽ xây dựng một cây phân tích cho một biểu thức số học từ dữ liệu nhập là một biểu thức dạng postfix. Đây là một hiệu chỉnh đơn giản của chương trình trong chương trước dùng để lượng giá các biểu thức postfix bằng cách dùng một ngăn xếp (stack). Thay vì lưu các kết quả trung gian của các tính toán trên ngăn xếp thì chúng ta lưu các cây biểu thức như trong cài đặt sau đây:

```
type link=↑node;
   node=record info:char; l,r:link end;
var x,z:link; c:char;
begin
stackinit;
new(z); z↑.l:=z; z↑.r:=z;
repeat
   repeat read(c) until c=' ';
   new(x); x↑.info:=c;
   if (c='*') or (c='+')
      then begin x↑.r:=pop; x↑.l:=pop end
      else begin x↑.r:=z; x↑.l:=z end;
   push(x);
until eoln;
```

Các thủ tục stackinit, push, và pop ở đây liên quan đến đoạn chương trình trong chương 3, được sửa đổi bằng cách đặt các **liên kết** vào ngăn xếp thay vì đặt các số nguyên vào đó. Các thủ tục này sẽ không được trình bày ở đây. Mỗi nút có một ký tự và hai liên kết tới các nút khác. Mỗi khi gặp một ký tự mới khác khoảng trống thì một nút được tạo ra bằng cách dùng thủ tục new của Pascal. Nếu nó là một toán tử thì các cây con cho các toán hạng của nó được đặt vào đỉnh của ngăn xếp là null. Thay vì dùng các liên kết null như với các danh sách, chúng ta dùng một nút nháp là z mà các liên kết của nó chỉ tới chính nó. Trong chương 14, chúng ta sẽ kiểm tra chi tiết để xem điều này làm các thao tác trên cây thuận tiện hơn như thế nào. Hình 4.5 cho thấy các trạng thái trung gian trong quá trình xây dựng cây trong hình 4.4. Chương trình đơn giản có thể được sửa đổi để phù hợp với các biểu thức phức tạp hơn bao gồm các toán tử đối số đơn như các biểu thức lũy thừa. Tuy nhiên sơ đồ rất tổng quát: một sơ đồ y hệt được dùng ngay cả khi phân tích và biên dịch các chương trình Pascal. Sau khi cây phân tích được tạo lập thì nó có thể được dùng cho nhiều việc, chẳng hạn như lượng giá biểu thức hay tạo các chương trình máy tính để lượng giá biểu thức. Chương 21 sẽ thảo luận về các thủ tục tổng quát để xây dựng các cây phân tích. Dưới đây chúng ta sẽ thấy cây có thể được dùng như thế nào để lượng giá biểu thức. Tuy nhiên, với mục đích của chương này, chúng ta thú vị nhất trong các sơ đồ của sự xây dựng cây.



Hình 4.5 *Xây dựng cây phân tích cho biểu thức ABD+DE**F+**

Như với các danh sách liên kết, luôn luôn có thể thay thế các mảng song song cho các con trỏ và mẩu tin để cài đặt cấu trúc dữ liệu cây nhị phân. Như trước đây, điều này đặc biệt hữu dụng khi số nút được biết trước, cũng như vậy, trường hợp đặc biệt cụ thể khi các nút chiếm một mảng sẽ được dùng cho mục đích khác.

Biểu diễn hai liên kết cho cây nhị phân đã dùng ở trên cho phép duyệt xuống phía dưới cây nhưng không cung cấp phương pháp duyệt lên phía trên cây. Tình huống này tương tự như các danh sách liên kết đơn so với các danh sách liên kết kép: người ta có thể thêm một liên kết khác vào mỗi nút để cho phép di chuyển tự do hơn, nhưng giá phải trả là sự cài đặt sẽ phức tạp hơn. Các lựa chọn đa dạng khác thì có sẵn trong các cấu trúc dữ liệu để làm giảm nhẹ sự di chuyển xung quanh cây, nhưng với các thuật toán trong quyển sách này thì sự biểu diễn hai liên kết đã đủ tổng quát.

Trong chương trình trên chúng ta đã dùng nút "nháp" thay thế cho các nút ngoài cũng giống như các danh sách liên kết, điều này trở nên thuận tiện hơn trong hầu hết các tình huống, nhưng không phải nó luôn luôn thích hợp, và có hai cách giải quyết khác thường dùng. Một cách là dùng một dạng nút khác cho các nút ngoài, dùng nút không có các liên kết. Phương pháp khác là đánh dấu các liên kết bằng một phương pháp nào đó (để phân biệt chúng với các liên kết khác ở trong cây), kế đến chúng chỉ đến nơi khác trong cây; một trong hai cách này sẽ được thảo luận bên dưới. Chúng ta sẽ gặp lại vấn đề này trong chương 14 và chương 17.

BIỂU DIỄN RỪNG

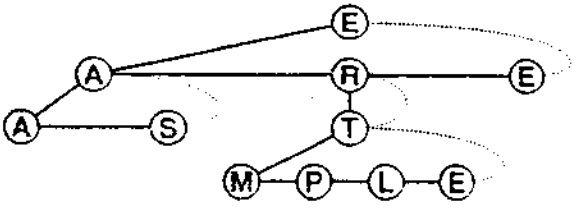
Các cây nhị phân có hai liên kết bên dưới mỗi nút trong, vì vậy biểu diễn được dùng ở trên là tự nhiên nhất. Nhưng chúng ta biểu diễn như thế nào đối với các cây tổng quát hay các rừng mà trong đó mỗi nút có thể đòi hỏi một số tùy ý các liên kết đến các nút bên dưới?

k	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*
a[k]	Ⓐ	Ⓢ	Ⓐ	Ⓜ	Ⓟ	Ⓛ	Ⓔ	Ⓣ	Ⓡ	Ⓔ	Ⓔ
dad[k]	*3*	*3*	*10*	*8*	*3*	*3*	*3*	*9*	*10*	*10*	*10*

Hình 4.6 *Biểu diễn liên kết cha của một cây*

Trước tiên, trong nhiều ứng dụng, chúng ta không cần duyệt xuống phía dưới cây mà chỉ duyệt lên! Trong các trường hợp như thế, chúng ta chỉ cần một liên kết tới cha của nó cho mỗi nút. Hình 4.6 cho thấy biểu diễn theo phương pháp này cho cây trong Hình 4.1: mảng a chứa thông tin kết hợp với mỗi mẩu tin và mảng dad chứa các liên kết cha. Do đó thông tin kết hợp với cha của a[i] là a[dad[i]]... Do quy ước gốc sẽ trỏ đến chính nó. Đây là một biểu diễn nén mà được giới thiệu nếu việc duyệt hướng lên phía trên cây là thích hợp. Chúng ta sẽ thấy ví dụ về sử dụng của biểu diễn này trong chương 22 và chương 30.

Để biểu diễn một rừng cho xử lý từ trên xuống, chúng ta cần một phương pháp để theo dõi con của mỗi nút mà không phải cấp phát trước một số cụ thể cho bất kỳ nút nào. Nhưng điều này chính là dạng thu hẹp mà các danh sách liên kết được thiết kế để thực hiện được thao tác xóa. Rõ ràng chúng ta nên dùng một danh sách liên kết cho các con của mỗi nút. Kế đến mỗi nút chứa hai liên kết, lại dùng cho danh sách liên kết của các con nó. Hình 4.7 minh họa biểu diễn này cho cây trong hình 4.1. Thay vì dùng một nút nhấp để kết thúc mỗi danh sách, chúng ta chỉ cần làm cho nó trỏ trở về cha của nó; điều này cung cấp một cách để di chuyển lên trên cũng như xuống dưới đây (Những liên kết này có thể được đánh dấu để phân biệt chúng với các liên kết "anh em"; chúng ta

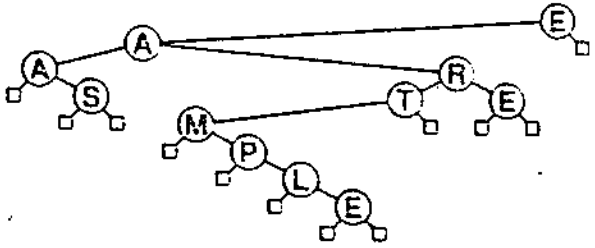


Hình 4.7 *Biểu diễn nút con trái nhất, và anh em bên phải của một cây*

cũng có thể quét xuyên qua các con của một nút bằng cách đánh dấu hay nhớ tên của cha sao cho quá trình quét có thể dừng khi gặp trở lại cha của chúng.)

Tuy nhiên trong biểu diễn này thì mỗi nút có chính xác hai liên kết (một tới các anh em bên phải, liên kết còn lại tới con trái nhất của nó). Người ta có thể thắc mắc không biết có sự khác nhau giữa cấu trúc dữ liệu này và cây nhị phân hay không. Câu trả lời là không có, xem hình 4.8 để thấy biểu diễn cây nhị phân của cây trong hình 4.1. Điều này cho thấy rằng bất kỳ rừng nào đều cũng có thể được biểu diễn như một cây nhị phân bằng cách làm cho liên kết trái của mỗi nút trỏ tới con trái nhất của nó, và liên kết phải của mỗi nút trỏ tới các anh em bên phải của nó (kết quả này thường làm ngạc nhiên các độc giả lần đầu tiên biết tới).

Từ đó chúng ta không ngại sử dụng rừng bất cứ lúc nào thuận tiện trong thiết kế thuật toán. Khi làm việc từ dưới lên trên, biểu diễn liên kết cha làm cho rừng được thao tác dễ dàng hơn, và khi làm việc từ trên xuống thì chúng tương đương thật sự với các cây nhị phân.



Hình 4.8 *Biểu diễn cây nhị phân của một cây*

DUYỆT CÂY

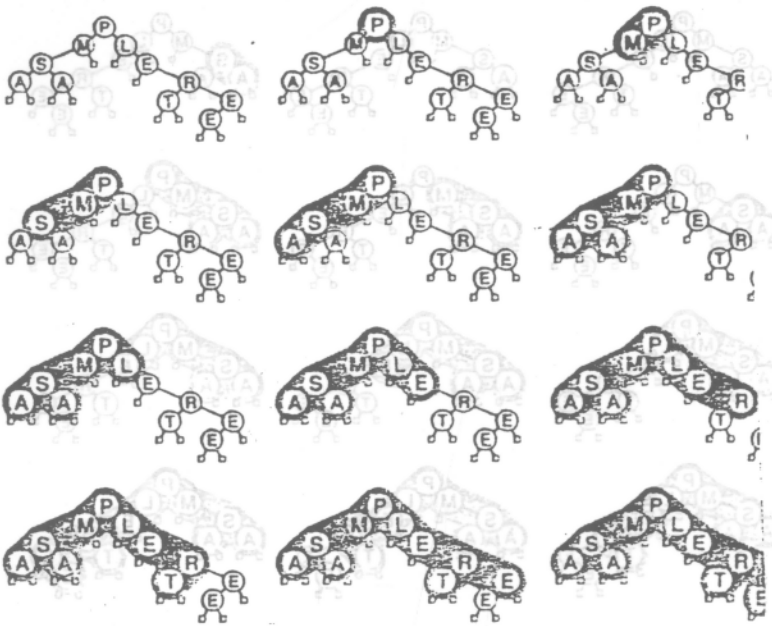
Mỗi khi một cây đã được tạo xong, việc đầu tiên người ta cần là làm thế nào để duyệt khắp mọi nút của nó: làm thế nào để "thăm" một cách hệ thống mọi nút. Thao tác này thì lại tầm thường đối với các danh sách tuyến tính nhờ vào định nghĩa của chúng, nhưng đối với cây thì có một số điểm khác trong quá trình xử lý. Các phương pháp khác nhau chủ yếu ở thứ tự mà chúng thăm các nút. Như chúng ta sẽ thấy, thứ tự nút khác nhau thì thích hợp cho các ứng dụng khác nhau.

Bây giờ chúng ta sẽ tập trung vào việc duyệt cây nhị phân. Bởi vì sự tương đương của các rừng và các cây nhị phân, các phương pháp được trình bày cũng hữu dụng cho rừng, nhưng chúng ta sẽ chú ý sau là làm thế nào mà áp dụng trực tiếp các phương pháp này vào rừng.

Phương pháp đầu tiên được khảo sát là duyệt theo **thứ tự đầu** (preorder traversal), như để viết ra biểu thức được biểu diễn bởi cây trong hình 4.4 dưới dạng prefix. Phương pháp này được định nghĩa đơn giản bằng đệ qui như sau: "thăm gốc, kế đó thăm cây con trái, kế đó thăm cây con phải". Cài đặt đơn giản nhất của phương pháp này là một cài đặt đệ qui sẽ được cho thấy trong chương tới và rất giống với cài đặt dựa vào ngăn xếp như sau:

```
procedure traverse(t:link);
begin
   push(t);
   repeat
      t:=pop; visit(t);
      if t<>z then push(t↑.r);
      if t<>z then push(t↑.l);
   until stackempty;
end;
```

(Ngăn xếp giả sử được khởi động bên ngoài thủ tục này). Theo quy luật này, chúng ta "thăm một cây con" bằng cách thăm gốc trước tiên. Kế đến, bởi vì chúng ta không thể thăm cả hai cây con một lúc, chúng ta phải lưu cây con phải vào ngăn xếp và thăm cây con trái. Khi cây con trái đã được thăm xong, cây con phải sẽ ở đỉnh của ngăn xếp; kế đến ta có thể thăm nó. Hình 4.9 cho thấy

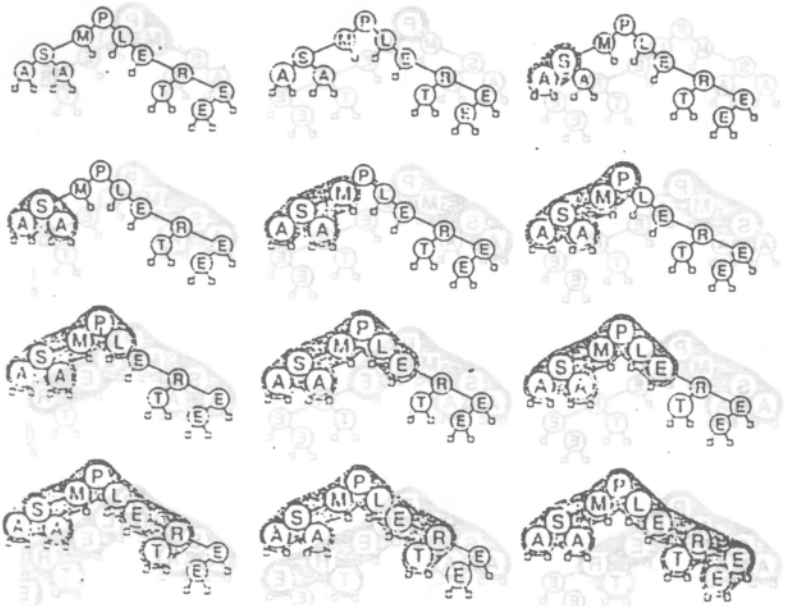


Hình 4.9 *Duyệt tiền thứ tự (Preorder traversal)*

chương trình này hoạt động khi áp dụng vào cây nhị phân trong hình 4.2: thứ tự mà các nút được thăm là P M S A A L E R T E E.

Để chứng minh chương trình này luôn thăm các nút theo thứ tự đầu, người ta có thể qui nạp với giả thiết qui nạp rằng các cây con đã được thăm theo các tiền thứ tự và nội dung của ngăn xếp ngay trước khi thăm một cây con thì y hệt như nội dung của nó ngay sau khi thăm.

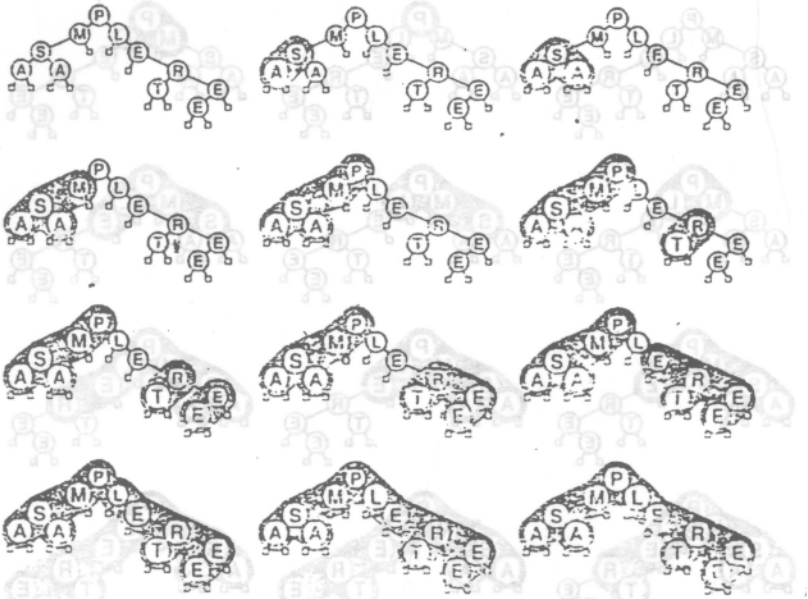
Phương pháp thứ hai là duyệt theo **thứ tự giữa** (inorder traversal), ví dụ như để viết một ra các biểu thức số học tương ứng với các cây phân tích dưới dạng infix (với vài công việc phụ để lấy đúng các dấu ngoặc). Tương tự như với cách duyệt theo thứ tự đầu, phương pháp này được định nghĩa bằng quy luật đệ qui như sau: "thăm cây con trái, kế đến thăm gốc, rồi thăm cây con phải".



Hình 4.10 *Duyệt trung thứ tự*

Phương pháp này đôi khi cũng được gọi là thứ tự đối xứng (symmetric order) với lý do hiển nhiên. Sự cài đặt chương trình dựa ngăn xếp cho phương pháp này thì hầu như giống với chương trình ở trên; chúng ta sẽ bỏ qua ở đây bởi vì nó sẽ là vấn đề chính của chương tới. Hình 4.10 cho thấy các nút của cây trong hình 4.2 được thăm như thế nào khi dùng phương pháp duyệt theo thứ tự giữa : các nút được thăm theo thứ tự A S A M P L E T R E E. Phương pháp này có khả năng được dùng rộng rãi nhất, ví dụ nó đóng vai trò trung tâm trong các ứng dụng của chương 14 và chương 15.

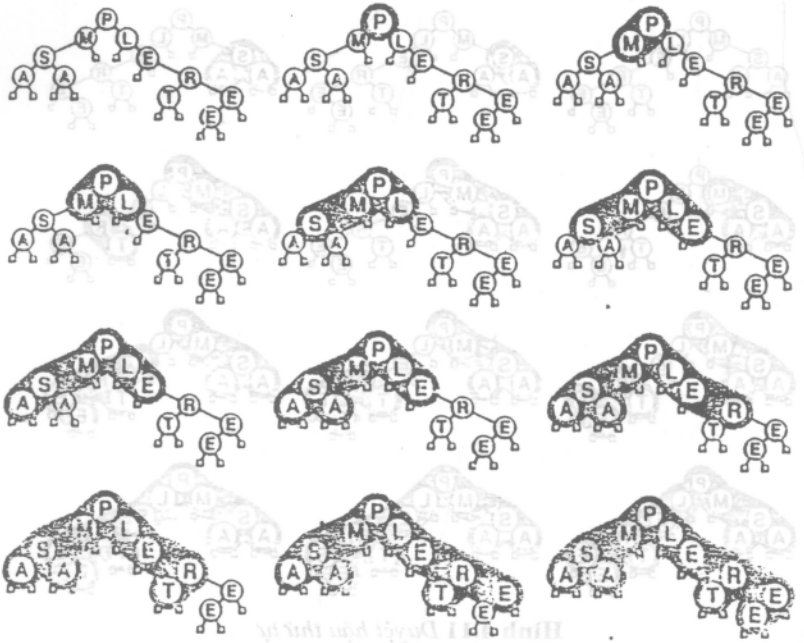
Dạng thứ ba của sự duyệt cây đệ qui được gọi là **thứ tự cuối** (postorder) được định nghĩa là "viếng thăm cây con trái, viếng thăm cây con phải, rồi thăm gốc". Hình 4.11 cho thấy các nút của cây trong hình 4.2 được thăm như thế nào khi dùng phương pháp



Hình 4.11 *Duyệt hậu thứ tự*

thứ tự cuối: các nút được thăm theo thứ tự A A S M T E E R E L P. Khi viếng thăm cây biểu thức của hình 4.4 bằng phương pháp này ta được biểu thức đung như mong đợi là ABC + DE * * F + *. Sự cài đặt chương trình dựa vào ngăn xếp cho phương pháp này phức tạp hơn hai phương pháp trên bởi vì phải dàn xếp cho gốc và cây con phải được lưu trong khi cây con trái được thăm và gốc phải được lưu trong khi cây con nhỏ được thăm. Các chi tiết của cài đặt này được dành như một bài tập cho độc giả.

```
procedure traverse(t:link);
begin
   put(t);
   repeat t:=get; visit(t);
      if t<>z then put(t↑.r);
      if t<>z then put(t↑.l);
   until queuempty;
end;
```



Hình 4.12 *Duyệt theo thứ tự tầng*

Chiến lược duyệt thứ tư mà chúng ta sẽ khảo sát thì hoàn toàn không đệ qui, chúng ta chỉ đơn giản viếng các nút như chúng xuất hiện trên giấy, đọc xuống từ đỉnh xuống dưới đáy và từ trái sang phải. Đây gọi là duyệt theo **thứ tự tầng** (level-order traversal) bởi vì tất cả các nút xuất hiện lần lượt xuất hiện trên mỗi tầng. Hình 4.12 cho thấy các nút của cây trong hình 4.2 được thăm theo thứ tự tầng như thế nào.

Điều đáng lưu ý là quá trình duyệt theo thứ tự tầng có thể cài đặt bằng cách dùng chương trình để duyệt theo thứ tự đầu với điều kiện dùng cấu trúc hàng đợi thay cho ngăn xếp:

Nói một cách khác thì chương trình này đồng nhất với chương trình trước và chỉ khác là dùng một cấu trúc dữ liệu FIFO thay vì dùng cấu trúc dữ liệu LIFO. Những chương trình này rất đáng được nghiên cứu kỹ lưỡng vì chúng đã nêu bật bản chất khác nhau

giữa ngăn xếp và hàng đợi. Chúng ta sẽ trở lại vấn đề này trong chương 30.

Các phương pháp duyệt cây theo thứ tự đầu, thứ tự cuối và thứ tự tầng thì cũng có thể được định nghĩa tương tự cho rừng. Để làm điều này, chúng ta hãy xem như rừng là một cây với một gốc tưởng tượng. Khi đó phương pháp thứ tự cuối là "thăm mỗi cây con, kế đến thăm gốc", phương pháp thứ tự đầu là "thăm gốc, kế đến thăm mỗi cây con", phương pháp thứ tự tầng thì y hệt như đối với cây nhị phân. Chú ý rằng phương pháp duyệt thứ tự đầu cho một rừng thì giống như phương pháp thứ tự đó cho cây nhị phân tương ứng như được định nghĩa ở trên, còn phương pháp thứ tự cuối cho một rừng giống như phương pháp thứ tự đó cho cây nhị phân, nhưng với thứ tự tầng thì không giống. Sự cài đặt trực tiếp dùng các ngăn xếp và hàng đợi là các khái quát đơn giản của các chương trình cho các cây nhị phân.

BÀI TẬP

1. Cho biết thứ tự mà các nút được thăm khi áp dụng các phương pháp duyệt theo thứ tự đầu, thứ tự giữa, thứ tự cuối, và thứ tự tầng đối với cây trong hình 4.3.

2. Cho biết độ cao của cây 4-hướng đầy đủ có N nút.

3. Vẽ cây phân tích của biểu thức (A+B) * C + (D+E)

4. Khảo sát cây trong hình 4.2 như một rừng nghĩa mà được biểu diễn như một cây nhị phân. Vẽ ra biểu diễn đó.

5. Cho biết nội dung của ngăn xếp mỗi khi một nút được thăm trong suốt quá trình duyệt theo thứ tự tầng được mô tả trong hình 4.9.

6. Cho biết nội dung của hàng đợi mỗi khi một nút được thăm trong suốt quá trình duyệt theo thứ tự tầng được mô tả trong hình 4.12.

7. Cho ví dụ về một cây mà ngăn xếp khi duyệt theo thứ tự đầu chiếm nhiều không gian lưu trữ hơn hàng đợi trong duyệt theo thứ tự tầng.

8. Cho ví dụ về một cây mà ngăn xếp khi duyệt theo thứ tự đầu chiếm ít không gian lưu trữ hơn hàng đợi trong duyệt theo thứ tự tầng.

9. Cho một cài đặt dựa trên ngăn xếp của phương pháp duyệt theo thứ tự cuối cho một cây nhị phân.

10. Viết một chương trình để cài đặt phương pháp duyệt theo thứ tự tầng của một rừng được biểu diễn như một cây nhị phân.

5
ĐỆ QUI

Đệ qui là một khái niệm cơ bản trong toán học và khoa học máy tính. Định nghĩa đơn giản của một chương trình đệ qui là chương trình mà gọi đến chính nó (và một hàm đệ qui là một hàm mà được định nghĩa dựa vào chính nó). Một chương trình đệ qui thì không thể gọi đến chính nó mãi mãi trừ khi nó không bao giờ dừng (và một hàm đệ qui không thể lúc nào cũng định nghĩa dựa vào chính nó trừ khi định nghĩa bị xoay vòng); một điều kiện cần thiết là phải có điều kiện kết thúc khi chương trình không gọi đến chính nó nữa (và khi hàm không được định nghĩa dựa vào chính nó). Tất cả những tính toán thực tế đều có thể diễn đạt bằng khuôn mẫu đệ qui.

Mục đích chính của chúng ta trong chương này là khảo sát đệ qui như một công cụ thực hành. Trước tiên chúng ta đưa ra vài ví dụ không thực tế để cho thấy mối liên hệ giữa qui nạp toán học và chương trình đệ qui. Kế đến chúng ta minh họa một ví dụ mẫu đầu tiên về một chương trình đệ qui "chia ra để trị", đây là dạng mà chúng ta sẽ dùng để giải các bài toán cơ bản trong những phần sau của quyển sách này. Cuối cùng chúng ta thảo luận làm thế nào để khử bỏ đệ qui từ một chương trình đệ qui bất kỳ và đưa ra một ví dụ chi tiết về sự khử đệ qui từ một thuật toán duyệt cây đệ qui đơn giản để đưa về một thuật toán dựa ngăn xếp không đệ qui.

Như chúng ta sẽ thấy, nhiều thuật toán lý thú hoàn toàn được giải quyết đơn giản bằng các chương trình đệ qui, và nhiều nhà thiết kế thuật toán thích diễn đạt vấn đề theo kiểu đệ qui. Nhưng cũng thường xảy ra trường hợp những thuật toán thú vị mà khó thấy các chi tiết của một cài đặt không đệ qui - trong chương này chúng ta sẽ thảo luận các kỹ thuật để tìm thấy những thuật toán như thế.

CÁC DÃY TRUY HỒI

Định nghĩa đệ qui của các hàm số rất thường xảy ra trong toán học, dạng đơn giản nhất mà bao gồm các đối số nguyên được gọi là các **quan hệ truy hồi**. Có lẽ hàm quen thuộc nhất thuộc loại này là hàm giai thừa, nó được định nghĩa bởi công thức:

$N! = N \cdot (N-1)!$ với $N >= 1$ và $0! = 1$

Hàm này tương ứng với chương trình đệ qui đơn giản sau đây:

```
function factorial(N:integer):integer;
begin
   if N=0
   then factorial:=1
   else factorial:=N*factorial(N-1);
end;
```

Chương trình minh họa các chức năng cơ sở của một chương trình đệ qui: nó gọi đến chính nó (với một giá trị đối số nhỏ hơn), và nó có một điều kiện kết thúc mà tính trực tiếp kết quả của nó. Mặt khác chúng ta thấy không có khó khăn gì khi cài đặt chương trình trên bằng vòng lặp for, vì vậy ví dụ này rất khó thuyết phục sức mạnh của đệ qui. Một điểm cần chú ý là nếu gọi chương trình trên với đối số âm, chẳng hạn factorial(-1) thì sẽ bị một vòng lặp vô hạn; đây là một lỗi thường xảy ra mà có thể xuất hiện tinh vi hơn đối với các chương trình đệ qui phức tạp.

Một quan hệ truy hồi nổi tiếng thứ hai là các số Fibonacci:

$F_N = F_{N-1} + F_{N-2}$ với $N=2$ và $F_0=F_1=1$

Quan hệ này định nghĩa dãy

1,1,2,3,5,8,13,21,34,55,89,144,233,377,610,...

Một lần nữa chúng ta có chương trình đệ qui đơn giản sau đây:

```
function fibonacci(N:integer):integer;
begin
   if N<=1
   then fibonacci:=1
   else fibonacci:=fibonacci(N-1)*fibonacci(N-2);
end;
```

Ngay cả các ví dụ này cũng ít thuyết phục về "sức mạnh" của đệ qui; thay vì vậy nó cho thấy rằng đệ qui không nên được dùng một cách mù quáng hay không thể lường trước hậu quả. Vấn đề ở đây là việc gọi đệ qui cho thấy F_{N-1} và F_{N-2} được tính một cách độc lập, trong khi thật ra để tính F_{N-1} người ta có thể dùng F_{N-2} (và F_{N-3}). Có thể tìm được chính xác số lần gọi hàm fibonacci khi cần tính toán F_N: số lần gọi cần để tính F_N là số lần gọi để tính F_{N-1} cộng với số lần gọi để tính F_{N-2} trừ khi N=0 hay N=1 thì chỉ cần một lần gọi. Do đó số lần để tính F_N bằng đúng F_N, nhưng ta đã biết F_N khoảng a^N trong đó a # 1.61803... là "tỷ số vàng": thật sự đáng sợ khi chương trình trên là một thuật toán thời gian lũy thừa mà chỉ để tính các số Fibonacci ! Ngược lại vô cùng dễ tính toán F_N với thời gian tuyến tính như sau:

```
procedure fibonacci;
const max=25;
var i:integer; F:array[0..max] of integer;
begin
   F[0]:=0; F[1]:=1;
   for i:=2 to max do F[i]:=F[i-1]+F[i-2]
end;
```

Chương trình này tính max số Fibonacci đầu tiên, nó sử dụng một mảng max phần tử. (Bởi vì các số tăng lên theo lũy thừa nên max sẽ nhỏ.)

Thật ra kỹ thuật dùng mảng để lưu các kết quả trước đó là phương pháp thường được chọn để tính toán các quan hệ truy hồi thường xuất hiện khi chúng ta muốn xác định tính năng của các chương trình đệ qui, chúng ta sẽ thấy qua nhiều ví dụ của quyển sách này. Ví dụ như trong chương 9 chúng ta sẽ gặp phương trình:

$$C_N = N-1 + \frac{1}{n} \sum_{k=1..N} (C_{k-1} + C_{N-k}) \text{ với N>=1 và } C_0=1.$$

Giá trị của C_N có thể được tính dễ dàng hơn bằng cách dùng một mảng như trong chương trình trên. Trong chương 9 chúng ta sẽ thảo luận làm thế nào để sử dụng công thức này, và nhiều quan hệ truy hồi thường sử dụng sẽ thảo luận trong chương 6.

Mối quan hệ giữa các chương trình đệ qui và các hàm được định nghĩa đệ qui thì có ý nghĩa triết học hơn là ý nghĩa thực tế. Nói một cách chính xác, vấn đề được đưa ra ở trên không những muốn bàn đến chính khái niệm đệ qui mà còn muốn bàn đến vấn đề cài đặt: một chương trình biên dịch thông minh có thể khám phá ra rằng hàm giai thừa có thể được cài đặt hiệu quả bằng vòng lặp và hàm Fibonacci có thể được sử dụng tốt hơn bằng cách lưu tất cả các giá trị được tính trước đó vào một trong mảng. Dưới đây chúng ta sẽ quan sát chi tiết hơn về sơ đồ cài đặt các chương trình đệ qui.

CHIA ĐỂ TRỊ

Hầu hết các chương trình đệ qui mà chúng ta khảo sát trong quyển sách này đều sử dụng hai lần gọi đệ qui, mỗi lần dùng một nữa dữ liệu nhập. Vấn đề này cũng được gọi là sơ đồ "chia để trị" trong lĩnh vực thiết kế thuật toán và đạt được lợi ít về kinh tế đáng kể. Các chương trình chia để trị thông thường không thể đưa về các vòng lặp bình thường giống như chương trình tính giai thừa ở trên, bởi vì chúng có hai lần gọi đệ qui; chúng cũng không thể đưa về dạng giống như chương trình tính các số Fibonacci, bởi vì dữ liệu nhập được chia làm đôi nhưng không có phần chung.

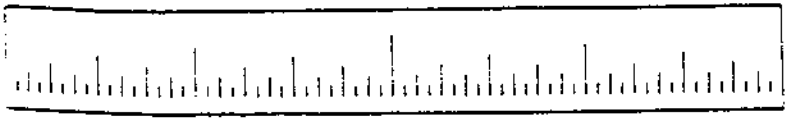
Bây giờ chúng ta hãy xét ví dụ vẽ các nét vạch cho mỗi inch trên một cây thước: có một nét vạch ở điểm 1/2", các nét vạch ngắn hơn một ít ở các đoạn 1/4", các nét vạch ngắn hơn nữa ởcác đoạn 1/8"... như trong hình 5.1 (dưới dạng phóng đại). Như chúng sẽ thấy có nhiều phương pháp giải cho ví dụ này, và đây là một ví dụ đơn giản về các tính toán theo kiểu chia để trị.

Nếu độ phân giải cần thiết là $1/2^n$", để cho đơn giản chúng ta nhân thêm một hệ số 2^n, vì vậy chúng ta phải đặt các nét vạch ở mỗi điểm giữa 0 và 2^n, không cần để ý đến các điểm đầu mút. Giả sử có sẵn một thủ tục mark(x,h) để vẽ một nét vạch có chiều cao h đơn vị tại vị trí x. Nét vạch giữa cao n đơn vị, các nét vạch giữa hai đoạn con sẽ có chiều cao n-1 đơn vị, ... Chương trình đệ qui "chia để trị" sau đây để giải quyết hoàn chỉnh vấn đề của chúng ta.

```
procedure rule(l,r,h:integer);
var m:integer;
   begin
      if h>0 then
      begin
         m:=(l+r) div 2; mark(m,h);
         rule(l,m,h-1); rule(m,r,h-1);
      end;
   end;
```

Ví dụ nếu gọi rule(0,64,6) thì sẽ được kết quả như hình 5.1, với tỷ lệ thích hợp. Ý tưởng của phương pháp nói trên là: để đánh dấu các nét vạch trong một đoạn, trước tiên ta vẽ một nét vạch dài ở giữa, thao tác này chia đoạn làm hai nửa bằng nhau, kến đến vẽ các vạch (ngắn hơn) trong một nửa đầu tiên bằng cùng một thủ tục, kế đến vẽ các vạch (ngắn hơn) trong một nửa thứ hai bằng cùng một thủ tục như những lần vừa vẽ.

Hãy chú ý vào điều kiện kết thúc của chương trình đệ qui bởi vì nếu không thì có thể chương trình sẽ không bao giờ dừng! Trong chương trình trên, chúng ta dừng khi độ dài của nét vạch là 0. Hình 5.2 minh họa chi tiết của quá trình xử lý, chúng ta thấy một danh sách các thủ tục được gọi và các vạch có được sau khi gọi thủ tục rule(0,8,3).



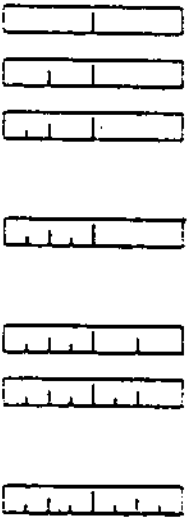
Hình 5.1 *Một cây thước*

Với bài tập này, ta thấy thứ tự các nét vẽ sẽ không thích hợp lắm, nếu như đặt dòng lệnh gọi thủ tục mark vào giữa hai dòng lệnh đệ qui thì ta sẽ có thứ tự khác hoàn chỉnh hơn như thấy trong hình 5.3 trang sau, trong đó các nét vẽ theo thứ tự từ trái sang phải.

Tập hợp các nét vẽ có được do hai thủ tục này là như nhau; nhưng thứ tự lại hoàn toàn khác nhau, điều này sẽ được giải thích bởi biểu đồ hình cây trong hình 5.4. Biểu đồ này có một nút tương

ứng với một lần gọi thủ tục rule, mỗi nút được gắn nhãn bởi các tham số khi gọi thủ tục, con của mỗi nút trong cây thì tương ứng với các lần gọi đệ qui cho thủ tục rule. Hình 5.2 tương ứng với duyệt cây này theo phương pháp tiền thứ tự (ở đây "viếng thăm" một nút là tương ứng với gọi tới thủ tục mark); Hình 5.3 tương ứng với duyệt cây theo phương pháp trung thứ tự.

Nói chung thì các thuật toán chia để trị hoạt động bằng cách tách dữ liệu nhập thành hai thành phần, hay phối hợp kết quả của hai bộ phận dữ liệu nhập được xử lý độc lập, hay trợ giúp một số việc sau khi một nữa dữ liệu nhập đã được xử lý. Nghĩa là có thể có code chương trình trước, sau, hay chen giữa hai lần gọi đệ qui. Chúng ta sẽ thấy nhiều ví dụ về các thuật toán như thế sau này, đặc biệt là trong các chương 9, 12, 27, 28, và 41. Chúng ta cũng gặp các thuật toán mà không thể hoàn toàn theo phương pháp chia để trị: có thể dữ liệu nhập được tách thành các mẩu không bằng nhau hai thành nhiều hơn hai mẩu, hay có thành phần chung giữa các mẩu. Trong chương kế tiếp, chúng ta sẽ nghiên cứu các quan



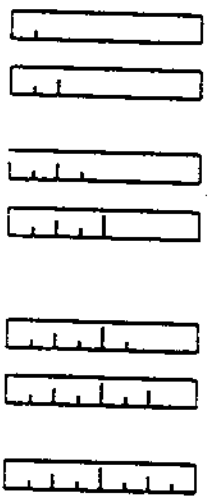
Hình 5.2 *Vẽ một cây thước*

hệ truy hồi giúp chúng ta xác định chính xác các chiến lược chia để trị có thể tiết kiệm bao nhiêu thời gian và không gian lưu trữ.

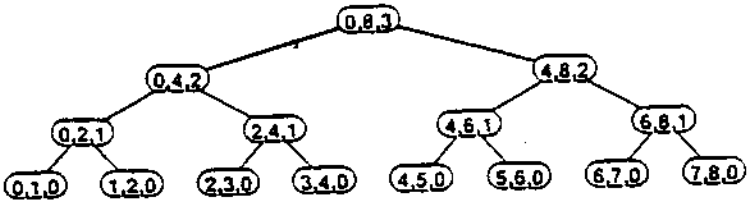
Chúng ta có thể dễ dàng xây dựng các thuật toán không đệ qui cho bài toán này. Phương pháp đơn giản nhất là vẽ chúng theo thứ tự như trong hình 5.3, bằng cách dùng một vòng lặp trực tiếp

for i:=1 **to** N-1 **do** mark (i,height(i)).

Hàm height(i) được tính không khó khăn lắm, chỉ cần dựa vào biểu diễn nhị phân của i. Như đã được chú ý trong chương 2 và được thảo luận chi tiết trong chương 10, sự làm việc với biểu diễn nhị phân của các số thì khả thi nhưng lại không phù hợp đối với ngôn ngữ Pascal. Có thể chuyển thuật toán đệ qui của bài toán này thành một thuật toán không đệ qui nhờ vào một qui trình "khử đệ qui" rất cầu kỳ mà chúng ta sẽ trình bày bên dưới cho một bài toán khác.



Hình 5.3 *Vẽ một cây thước (phiên bản trung thứ tự)*



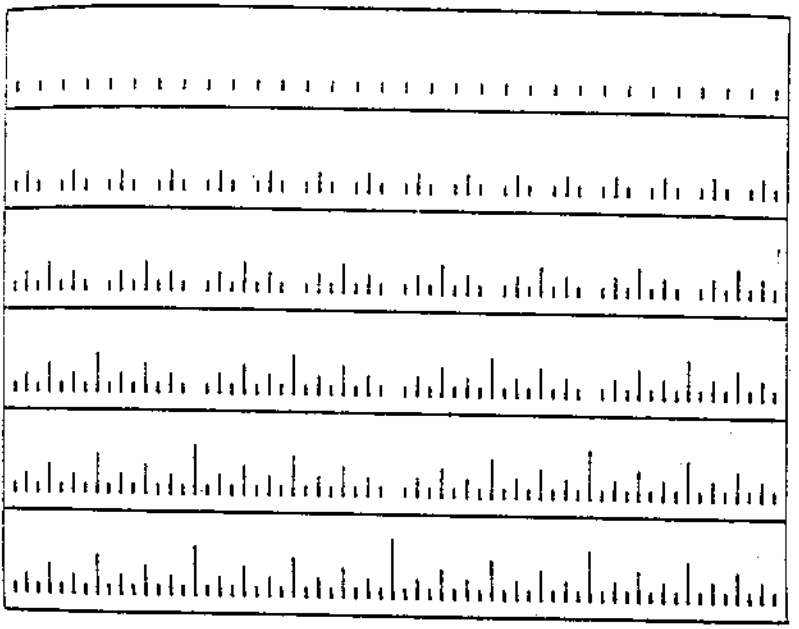
Hình 5.4 *Cây gọi đệ quy để vẽ một cây thước*

Một thuật toán không đệ qui khác mà không tương ứng với bất kỳ một cài đặt đệ qui nào là vẽ các nét vạch ngắn nhất trước tiên, kế đến vẽ các nét vạch dài hơn một ít... như trong chương trình gọn hơn sau đây:

```
procedure rule(l,r,h:integer);
var i,j:integer;
   begin j:=1;
         for i:=1 to h do
         begin for x:=0 to (l+r) div j do mark(l+j+x*(j+j),i);
               j:=j+j;
         end;
   end;
```

Hình 5.5 trang sau cho thấy chương trình này vẽ các nét vạch như thế nào, quá trình này tương ứng với việc duyệt cây trong hình 5.4 theo thứ tự tầng (từ dưới lên trên) mà không dùng đệ qui.

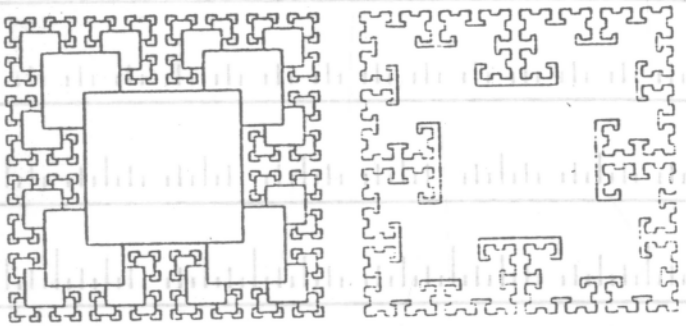
Thuật toán vừa nói tương ứng với một phương pháp tổng quát trong việc thiết kế thuật toán, chúng ta giải một bài toán bằng cách giải quyết trước các trường hợp tầm thường, kế đến phối hợp những lời giải này để giải quyết trường hợp tổng quát hơn một ít... cho đến khi toàn bộ bài toán được giải xong, cách tiếp cận này được gọi là "tổ hợp và giải quyết". Trong khi bất kỳ chương trình đệ qui nào đều có thể cài đặt lại một cách không đệ qui thì ngược lại không phải luôn luôn có thể bố trí lại các tính toán bằng phương pháp này. Nhiều chương trình đệ qui buộc các bài toán con được giải theo một thứ tự cụ thể. Cách tiếp cận này theo kiểu từ dưới lên, nó ngược lại cách tiếp cận từ trên xuống của phương pháp "chia ra để trị". Chúng ta sẽ gặp nhiều ví dụ về vấn đề này, quan trọng nhất là trong chương 12. Một sự tổng quát hóa của phương pháp này sẽ được thảo luận trong chương 42.



Hình 5.5 *Vẽ một cây thước không đệ quy*

Hình 5.6 cho thấy một mẫu hai chiều minh họa một mô tả đệ qui đơn giản mà lại dẫn tới một hình phức tạp. Chương trình vẽ ra mẫu bên trái chỉ là sự tổng quát hóa một ít của chương trình vẽ thước : hàm vẽ cơ bản được dùng trong chương trình chỉ là hàm vẽ một hình vuông kích thước 2r có tâm tại (x,y). Tương tự như mẫu bên trái, độc giả hãy cố gắng tìm một thuật toán đệ qui cho mẫu bên phải. Độc giả cũng nên tìm một thuật toán không đệ qui cho mẫu này.

Các mẫu hình học được định nghĩa đệ qui giống như hình 5.6 đôi khi cũng được gọi là các mẫu fractal. Nếu chúng ta dùng các hàm vẽ cơ bản phức tạp hơn và nhiều mô tả đệ qui phức tạp hơn (đặc biệt là các hàm được định nghĩa đệ qui trên mặt phẳng thực và phức) thì chúng ta sẽ có các mẫu phức tạp và đa dạng rất đáng quan tâm.



Hình 5.6 *Một ngôi sao fractal được vẽ với các hộp (bên trái) và một ngôi sao được vẽ bởi đường viền của các hộp (bên phải).*

```
procedure star(x,y,r:integer);
  begin
    if r0 then
    begin
      star(x-r,y+r,r div 2);
      star(x+r,y+r,r div 2);
      star(x-r,y-r,r div 2);
      star(x+r,y-r,r div 2);
      box(x,y,r);
    end
  end;
```

DUYỆT CÂY THEO PHƯƠNG PHÁP ĐỆ QUI

Như đã thấy trong chương 4, có lẽ phương pháp đơn giản nhất để duyệt các nút của một cây là sự cài đặt đệ qui. Ví dụ, chương trình sau đây sẽ viếng các nút của một cây nhị phân theo kiểu trung thứ tự:

```
procedure trawerse(t:link);
  begin if t<>z then begin trawerse(t↑.l); visit(t); trawerse(t↑.r); end;
  end;
```

Cài đặt này phản ánh chính xác định nghĩa của phương pháp duyệt trung thứ tự: "Nếu cây không trống thì trước tiên duyệt cây con bên trái, kế đến viếng thăm nút gốc, và kế đến là duyệt cây con bên phải". Hiển nhiên phương pháp tiền thứ tự cũng có thể được cài đặt bằng cách đặt dòng lệnh gọi thủ thủ tục visit ở phía dưới của hai dòng lệnh gọi đệ qui.

Sự cài đặt đệ qui cho bài toán duyệt cây thì tự nhiên hơn nhiều so với cài đặt dựa vào ngăn xếp bởi vì các cây đều là các cấu trúc được định nghĩa đệ qui và bởi vì các phương pháp duyệt tiền thứ tự, trung thứ tự, và hậu thứ tự cũng đều là các tiến trình được định nghĩa một cách đệ qui. Nhưng ngược lại chúng ta thấy rằng không có phương pháp thông thường để cài đặt một thủ tục đệ qui cho phương pháp duyệt cây theo thứ tự tầng. Chúng ta sẽ quay lại vấn đề này trong chương 29 và 30 khi khảo sát các thuật toán duyệt cho các đồ thị mà có các cấu trúc phức tạp hơn cây nhiều.

Sự sửa đổi đơn giản chương trình đệ qui nói trên và sự cài đặt thích hợp cho thủ visit có thể đưa tới các chương trình thao tác trên nhiều tính chất rất đa dạng của cây. Ví dụ như chương trình sau đây cho thấy làm cách nào để tính được tọa độ để đặt các nút của cây nhị phân trong các hình của quyển sách này. Giả sử mẩu tin được dùng để lưu trữ các nút bao gồm hai trường nguyên tương ứng với các tọa đô x và y của nút trên giấy. (Để tránh các chi tiết về sự tỉ lệ và tịnh tiến, những tọa độ này được giả sử là các tọa độ tương đối: nếu một cây có N nút và có chiều cao h, thì tọa x hướng trái tới phải từ 1 tới N và tọa độ y hướng trên xuống dưới từ 1 tới h.) chương trình sau đây đặt vào các trường này các giá trị thích hợp cho mỗi nút:

```
procedure visit(t:link);
   begin x:=x+1; t↑.x:=x; t↑.y:=y;
end;
procedure traverse(t:link);
   begin y:=y+1;
         if t<> z then begin traverse(t↑.l); visit(t); traverse(t↑.r); end;
         y:=y-1;
   end;
```

Chương trình dùng 2 biến toàn cục x và y được giả sử là khởi động giá trị 0. Biến x theo dõi số nút đã được viếng theo kiểu trung thứ tự, biến y theo dõi độ cao của cây. Mỗi lần thủ tục traverse duyệt xuống phía dưới cây thì biến y tăng lên 1 và mỗi lần thủ tục này duyệt xuống phía trên cây thì biến y giảm 1.

Với một cách làm tương tự như trên, người ta có thể cài đặt các chương trình đệ qui để tính chiều dài đường đi của cây, có thể cài đặt một phương pháp khác khác để vẽ một cây, để lượng giá một biểu thức được biểu diễn bởi cây.

KHỬ ĐỆ QUI

Có mối quan hệ gì giữa sự cài đặt đệ qui vừa nói trên và sự cài đặt không đệ qui trong chương 4 cho bài toán duyệt cây? chắc chắn rằng chúng liên hệ nhau rất nhiều, bởi vì với một cây bất kỳ thì hai chương trình cùng cho ra chính xác một dãy các lời gọi tới thủ tục visit. Trong phần này, chúng ta sẽ nghiên cứu chi tiết câu hỏi này nhờ vào sơ đồ khử bỏ đệ qui từ chương trình duyệt tiền thứ tự được cho ở trên để có được một cài đặt không đệ qui.

Điều này cũng y hệt như công việc mà một trình biên dịch phải làm khi dịch một chương trình đệ qui thành ngôn ngữ máy. Mục đích của chúng ta không chú trọng nghiên cứu các kỹ thuật biên dịch mà là chú trọng các mối quan hệ giữa các cài đặt đệ qui và không đệ qui của thuật toán.

Chúng ta khởi đầu với một cài đặt đệ qui của phương pháp duyệt cây theo kiểu tiền thứ tự chính xác như mô tả ở trên:

```
procedure traverse(t:link);
   begin
   if t <>z then begin visit(t); traverse(t↑.l); traverse(t↑.r); end;
   end;
```

Trước hết lệnh gọi đệ qui thứ hai có thể được khử dễ dàng bởi vì không có mã chương theo sau nó, khi lệnh này thực hiện thì thủ tục traverse được gọi với tham số t↑.r và khi lệnh gọi này kết thúc thì lần gọi thủ tục traverse hiện hành cũng kết thúc, chúng ta có thể dùng goto thay vì một lệnh gọi đệ qui như sau:

```
procedure traverse(t:link);
label 0,1;
   begin
      0: if t=z then goto 1 ;
      visit(t);
      traverse(t↑.l);
      t:=t↑.r;
      goto 0;
1:  end;
```

Đây là một kỹ thuật nổi tiếng được gọi là khử đệ qui phần cuối (end-recursion removal) và được cài đặt trên rất nhiều trình biên dịch. Việc khử lần gọi đệ qui còn lại đòi hỏi phải làm nhiều việc hơn. Nói chung hầu hết các trình biên dịch đều phát sinh mã chương trình cho bất kỳ một lời gọi thủ tục theo cách tương tự như sau: "Đặt tất cả các giá trị của các biến cục bộ và địa chỉ của chỉ thị kế tiếp vào ngăn xếp, qui định các giá trị tham số cho thủ tục và chuyển tới (goto) vị trí bắt đầu của thủ tục". Khi thủ tục hoàn tất thì nó phải "lấy ra khỏi ngăn xếp địa chỉ trả về và các giá trị của các biến cục bộ, khôi phục các biến và chuyển tới (goto) địa chỉ trả về." Dĩ nhiên rằng mọi việc sẽ phức tạp hơn đối với tình huống tổng quát mà một trình biên dịch thực sự phải giải quyết, nhưng chúng ta sẽ lợi dụng kỹ thuật vừa nói để khử lần đệ qui thứ hai khỏi chương trình của chúng ta, như ở trang sau:

Ở đây chỉ có một biến cục bộ là t nên chúng ta đặt nó vào ngăn xếp và chuyển tới (goto) vị trí bắt đầu; chỉ có một địa chỉ trả về là 3 nhưng lại là một số cố định nên chúng ta không cần đặt nó vào ngăn xếp. Ở cuối của thủ tục này chúng ta lấy lại giá trị cũ của t từ

ngăn xếp và chuuyển tới (goto) địa chỉ trả về 3. Khi ngăn xếp trống ta quay trở về từ lời gọi đầu tiên tới thủ tục traverse.

```
procedure traverse(t:link);
label 0,1,2,3;
   begin
   0: if t=z then goto 1;
   visit(t);
   push(t);
   t:=t↑.l;
   goto 0;
   3: t:=t↑.r; goto 0;
   1: if stackempty then
   goto 2;
   t:=pop;
   goto 3;
2: end;
```

Bây giờ thì đã khử được đệ qui nhưng lại có quá nhiều các chỉ thị **goto** làm cho chương trình rất lượm thượm, nhưng chúng ta cũng khử "một cách cơ học" để đưa đến một chương trình có cấu trúc hơn. Trước tiên đoạn chương trình giữa nhãn 3 và chỉ thị **goto 0** thứ hai được bao quanh bởi các chỉ thị goto nên có thể khử đi một cách đơn giản, khử đi nhãn 3 cùng với chỉ thị **goto**. Kế tới, đoạn chương trình giữa nhãn 0 và chỉ thị **goto 0** thứ nhất thì chẳng có gì khác với một vòng lặp **while**, và chúng ta có được chương trình sau:

```
procedure traverse(t:link);
label 0,2;
   begin
   0: while t<>z do
      begin
      visit(t);
      push(t↑.r);
      t:=t↑.l;
      end;
      if stackempty then goto 2;
      t:=pop;
      goto 0;
2: end;
```

Bây giờ chúng ta có một vòng lặp khác mà có thể chuyển thành một vòng lặp **repeat** bằng cách cho thêm một chỉ đặt vào ngăn xếp, và có được một chương trình không có **goto** :

```
procedure traverse(t:link);
  begin
    push(t);
    repeat
      t:=pop;
      while t<>z do
      begin visit(t); push(t↑.r); t:=t↑.l;
      end;
    until stackempty;
  end;
```

Chương trình này là phương pháp duyệt cây không đệ qui "chuẩn". Quá trình vừa thực hiện bên trên thật là một bài tập rất đáng giá cho chúng ta theo dõi.

Như thường lệ, cấu trúc vòng lặp bên trong một vòng lặp của chương trình nói trên có thể được đơn giản hóa hơn (và trả giá trị một vài chỉ thị đặt vào ngăn xếp):

```
procedure traverse(t:link);
  begin
  push(t);
  repeat
    t:=pop;
    if t<>z then
      begin
      visit(t);
      push(t↑.r);
      push(t↑.l);
      end;
  until stackempty;
  end;
```

Chương trình này tương tự như thuật toán duyệt cây tiền thứ tự đệ qui ban đầu của chúng ta, nhưng hai chương trình thực sự hoàn toàn khác nhau. Một khác nhau chính là chương trình này có thể chạy được trong bất kỳ môi trường lập trình nào, trong khi sự cài đặt đệ qui thì hiển nhiên đòi hỏi một môi trường cung cấp khả

năng gọi đệ qui. Ngay cả trong một môi trường như thế thì phương pháp cài đặt dựa ngăn xếp này cũng hiệu quả hơn.

Cuối cùng, chúng ta chú ý rằng chương trình này đặt các cây con NULL vào ngăn xếp, đó là một hậu quả trực tiếp của sự quyết định trong cài đặt ban đầu nhằm để kiểm tra xem cây con có NULL hay không trong lần gọi đệ qui đầu tiên. (Sự cài đặt đệ qui cũng có thể kiểm tra $t^{\uparrow}.l$ và $t^{\uparrow}.r$ để chỉ cần gọi đệ qui cho các cây con khác NULL.) Bằng cách thay đổi chương trình trên để tránh việc đặt các cây co NULL vào ngăn xếp ta có chương trình duyệt cây tiền thứ tự dựa ngăn xếp của chương 4.

```
procedure trawerse(t:link);
begin
   push(t);
   repeat
      t:=pop; visit(t);
      if t↑.rz then push(t↑.r);
      if t↑.lz then push(t↑.l);
   until stackempty;
end;
```

Bất kỳ thuật toán đệ qui nào cũng có thể làm như trên để khử bỏ đệ qui, thật ra đây là nhiệm vụ chính của các chương trình biên dịch. Sự khử đệ qui vừa mô tả ở trên, mặc dù phức tạp nhưng nó thường dẫn đến cả sự cài đặt không đệ qui hiệu quả hơn lẫn sự hiểu biết tốt hơn về tính chất tự nhiên của tính toán đệ qui.

BÀN LUẬN THÊM

Chắc chắn rằng chúng ta không thể đánh giá đúng đắn một chuyên đề cơ bản cỡ như sự đệ qui chỉ trong một thảo luận quá ngắn. Sẽ có nhiều ví dụ rất tốt về các chương trình đệ qui xuất hiện suốt trong quyển sách này, các thuật toán "chia để trị" đã và đang được áp dụng để giải quyết nhiều vấn đề đa dạng và rộng rãi. Với nhiều ứng dụng thì cài đặt đệ qui là phương pháp trực tiếp và đơn giản, đối với các ứng dụng khác chúng ta sẽ khảo sát tính hiệu quả của sự khử đệ qui được mô tả trong chương nầy hay là thay bởi một cài đặt trực tiếp không dùng đệ qui.

Đệ qui là quả tim của các nghiên cứu lý thuyết rất tự nhiên của tính toán. Các hàm và chương trình đệ qui đóng vai trò trung tâm trong các nghiên cứu toán học mà cố gắng tách rời các bài toán có thể giải quyết bằng máy tính khỏi các bài toán không thể giải quyết bằng máy tính.

Trong chương 44, chúng ta sẽ nghiên cứu sử dụng các chương trình đệ qui (và các kỹ thuật khác) để giải các bài toán khó mà cần phải chọn ra đáp số đúng từ một số lượng lớn các đáp số có thể có. Như chúng ta sẽ thấy, sự lập trình đệ qui có thể là phương tiện hoàn toàn hiệu quả để tổ chức một thao tác tìm kiếm phức tạp trong một tập hợp các khả năng có thể có.

BÀI TẬP

1. Hãy viết một chương trình đệ qui để vẽ một cây nhị phân sao cho nút gốc xuất hiện tại giữa trang giấy, gốc của cây con trái xuất hiện giữa của một nửa trái của trang giấy ...
2. Hãy viết một chương trình đệ qui để tính độ dài đường đi ngoài của một cây nhị phân.
3. Hãy viết một chương trình đệ qui để tính độ dài đường đi ngoài của một cây được biểu diễn như một cây nhị phân.
4. Hãy cho ra các tọa độ có được khi thủ tục đệ qui vẽ cây trong bài được áp dụng vào cây nhị phân hình 4.2.
5. Hãy khử bỏ đệ qui khỏi chương trình fibonacci trong bài để có một cài đặt không đệ qui.
6. Hãy khử bỏ đệ qui khỏi chương trình duyệt cây inorder trong bài để có một cài đặt không đệ qui.
7. Hãy khử bỏ đệ qui khỏi chương trình duyệt cây postorder trong bài để có một cài đặt không đệ qui.
8. Viết một chương trình đệ qui "chia để trị" để vẽ xấp xỉ đoạn thẳng nối hai điểm (x1, y1) và (x2, y2) bằng cách vẽ chỉ những điểm có tọa độ nguyên. (Hướng dẫn: trước tiên vẽ một điểm gần ở giữa.)
9. Hãy viết một chương trình đệ qui để giải bài toán Josephus (xem chương 3).
10. Hãy viết một cài đặt đệ qui của thuật toán Euclid (xem chương 1).

6
PHÂN TÍCH THUẬT TOÁN

Hầu hết các bài toán đều có nhiều thuật toán khác nhau để giải quyết chúng. Như vậy thì làm thế nào để chọn được sự cài đặt tốt nhất? Đây là một lĩnh vực được phát triển tốt trong nghiên cứu về khoa học máy tính. Chúng ta sẽ thường xuyên có cơ hội tiếp xúc với các kết quả nghiên cứu mà mô tả các tính năng của các thuật toán cơ bản. Tuy nhiên, việc so sánh các thuật toán thì rất cần thiết và chắc chắn rằng một vài dòng hướng dẫn tổng quát về phân tích thuật toán sẽ rất hữu dụng.

Thông thường các vấn đề mà chúng ta giải quyết có một "kích thước" tự nhiên (thường là số lượng dữ liệu được xử lý) mà chúng ta sẽ gọi là N. Chúng ta muốn mô tả tài nguyên cần được dùng (thông thường nhất là thời gian cần thiết) như một hàm số theo N. Chúng ta quan tâm đến **trường hợp trung bình**, tức là thời gian cần thiết để xử lý dữ liệu nhập thông thường, và cũng quan tâm đến **trường hợp xấu nhất**, tương ứng với thời gian cần thiết khi dữ liệu rơi vào trường hợp xấu nhất có thể có.

Vài thuật toán trong quyển sách này được hiểu rất tốt, theo nghĩa mà các công thức toán học đã cho biết thời gian chạy trung bình và thời gian chạy trong trường hợp xấu nhất. Các công thức như thế được phát triển cẩn thận nhờ vào nghiên cứu chương trình để tìm ra thời gian chạy dưới dạng các đại lượng toán học cơ bản và sẽ kèm theo một sự phân tích toán học của các đại lượng đó. Mặt khác, tính năng của một số thuật toán trong quyển sách này lại khó hiểu, có lẽ sự phân tích của chúng sẽ dẫn tới các câu hỏi toán học không thể giải đáp được, hay có lẽ các cài đặt đã biết thì quá phức tạp để có thể có một sự phân tích hợp lý cho chúng, hay thường xảy ra nhất là các dạng dữ liệu không thể được đặc trưng một cách đầy đủ. Hầu hết các thuật toán như thế đều rơi vào trường hợp: vài kết quả đã biết liên quan đến các tính năng của chúng, nhưng chúng không được phân tích thật đầy đủ.

Nhiều yếu tố quan trọng của việc phân tích thuật toán thường ra ngoài lĩnh vực ứng dụng của các lập trình viên. Trước hết, các chương trình Pascal được dịch thành mã máy cho một máy tính cụ thể nào đó, và có thể câu hỏi cần trả lời là phải tìm ra chính xác thời gian cần để thực hiện một lệnh Pascal (đặc biệt trong các môi trường mà tài nguyên được chia sẻ sao cho cùng một chương trình có thể có các đặc trưng tính năng bị thay đổi). Điều thứ hai là có nhiều chương trình vô cùng nhạy cảm với dữ liệu nhập của chúng và tính năng có chúng có thể phụ thuộc một cách không có qui luật vào dữ liệu nhập. Trường hợp trung bình có thể là một sự lý tưởng về mặt toán học mà lại không tiêu biểu cho dữ liệu thông thường mà chương trình đang được dùng, và trường hợp xấu nhất có thể là một cấu trúc kỳ dị không bao giờ xảy ra trong thực tế. Thứ ba là nhiều chương trình không thể được khảo sát tốt bởi vì không có sẵn các kết quả toán học cho nó. Cuối cùng là rất thường xảy ra trường hợp mà các chương trình không thể so sánh với nhau: một chương trình chạy rất hiệu quả trên một loại cụ thể của dữ liệu nhập, trong khi chương trình khác lại hiệu quả hơn đối với một tình huống khác.

Tuy nhiên các chú ý vừa đề cập cũng không không phải hoàn toàn cản trở chúng ta đánh giá các thuật toán, thông thường có thể tiên đoán chính xác một chương trình cụ thể cần bao nhiêu thời gian chạy, hay biết được rằng một chương trình sẽ hoạt động tốt hơn chương trình khác trong các hoàn cảnh cụ thể. Nhiệm vụ của người phân tích thuật toán là khám phá càng nhiều thông tin càng tốt về tính năng của thuật toán; và nhiệm vụ của thảo chương viên là nhờ vào các thông tin đó để chọn thuật toán cho các ứng dụng cụ thể. Trong chương nầy chúng ta quan tâm đến thế giới lý tưởng của người phân tích và trong chương tới chúng ta sẽ thảo luận các khảo sát thực tế về cài đặt.

KHUNG LÀM VIỆC

Bước đầu tiên trong việc phân tích một thuật toán là đặc trưng dữ liệu sẽ được dùng như dữ liệu nhập của thuật toán và quyết định phân tích nào là thích hợp. Về mặt lý tưởng, chúng ta muốn rằng với một phân bố tùy ý được cho của dữ liệu nhập thì sẽ có sự phân

bố tương ứng về thời gian hoạt động của thuật toán. Chúng ta không thể đạt tới điều lý tưởng nầy cho bất kỳ một thuật toán không tầm thường nào, vì vậy chúng ta chỉ quan tâm đến các thống kê về tính năng của thuật toán bằng cách cố gắng chứng minh thời gian chạy luôn luôn nhỏ hơn một "cận trên" **bất chấp dữ liệu** nhập như thế nào và cố gắng tính được thời gian chạy trung bình cho dữ liệu nhập "ngẫu nhiên".

Bước thứ hai trong phân tích một thuật toán là nhân ra các thao tác trừu tượng của thuật toán để tách biệt sự phân tích với sự cài đặt. Ví dụ, chúng ta tách biệt sự nghiên cứu có bao nhiêu phép so sánh trong một thuật toán sắp xếp khỏi sự xác định cần bao nhiêu micro giây trên một máy tính cụ thể; yếu tố thứ nhất được xác định bởi tính chất của thuật toán, yếu tố thứ hai lại được xác định bởi tính chất của máy tính. Sự tách biệt nầy cho phép chúng ta so sánh các thuật toán một cách độc lập với sự cài đặt cụ thể hay độc lập với một máy tính cụ thể.

Trong khi số các thao tác trừu tượng về nguyên tắc có thể rất lớn thì thông thường tính năng của các thuật toán mà chúng ta khảo sát chỉ phụ thuộc vào một vài đại lượng. Nói chung để nhận ra các đại lượng phù hợp cho một chương trình cụ thể thì người ta chọn một "phác họa" (có sẵn trong nhiều cài đặt Pascal) để cho biết tần số chỉ thị trong các lần chạy mẫu nào đó. Trong quyển sách nầy, chúng ta quan tâm tới các đại lượng quan trọng nhất thuộc loại đó cho mỗi chương trình.

Bước thứ ba trong quá trình phân tích thuật toán là sự phân tích về mặt toán học, với mục đích tìm ra các giá trị trung bình và trường hợp xấu nhất cho mỗi đại lượng cơ bản. Chúng ta sẽ không gặp khó khăn khi tìm một cận trên cho cho thời gian chạy chương trình, vấn đề ở chỗ là phải tìm ra chận trên tốt nhất, tức là cái mà đạt được khi gặp dữ liệu nhập của trường hợp xấu nhất. Trường hợp trung bình thông thường đòi hỏi một phân tích toán học tinh vi hơn trường hợp xấu nhất. Mỗi khi đã hoàn thành một quá trình phân tích thuật toán dựa vào các đại lượng cơ bản, nếu thời gian kết hợp với mỗi đại lượng được xác định rõ thì ta sẽ có các biểu thức để tính thời gian chạy.

Nói chung, tính năng của một thuật toán thường có thể được phân tích ở một mức độ rất chính xác, chỉ bị giới hạn bởi tính năng

không chắc chắn của máy tính hay bởi sự khó khăn trong việc xác định các tính chất toán học của một vài đại lượng trừu tượng. Tuy nhiên thay vì phân tích một cách chi tiết chúng ta thường thích ước lượng để tránh sa vào chi tiết.

SỰ PHÂN LỚP CÁC THUẬT TOÁN

Như đã được chú ý ở trên, hầu hết các thuật toán đều có một tham số chính là N, thông thường đó số lượng các phần tử dữ liệu được xử lý mà ảnh hưởng rất nhiều tới thời gian chạy. Tham số N có thể là bậc của một đa thức, kích thước của một tập tin được sắp xếp hay tìm kiếm, số nút trong một đồ thị .v.v... Hầu hết tất cả các thuật toán trong quyển sách nầy có thời gian chạy tiệm cận tới một trong các hàm sau:

1. Hầu hết các chỉ thị của các chương trình đều được thực hiện một lần hay nhiều nhất chỉ một vài lần. Nếu tất cả các chỉ thị của cùng một chương trình có tính chất nầy thì chúng ta sẽ nói rằng thời gian chạy của nó là hằng số. Điều nầy hiển nhiên là hoàn cảnh phấn đấu để đạt được trong việc thiết kế thuật toán.

2. logN Khi thời gian chạy của chương trình là **logarit**, tức là thời gian chạy chương trình tiến chậm khi N lớn dần. Thời gian chạy thuộc loại nầy xuất hiện trong các chương trình mà giải một bài toán lớn bằng cách chuyển nó thành một bài toán nhỏ hơn, bằng cách cắt bỏ kích thước bớt một hằng số nào đó. Với mục đích của chúng ta, thời gian chạy có được xem như nhỏ hơn một hằng số "lớn". Cơ số của logarit làm thay đổi hằng số đó nhưng không nhiều: khi N là một ngàn thì logN là 3 nếu cơ số là 10, là 10 nếu cơ số là 2; khi N là một triệu, logN được nhân gấp đôi. Bất cứ khi nào N được nhân đôi, logN tăng lên thêm một hằng số, nhưng logN không bị nhân gấp đôi tới khi tới khi N tăng tới N^2.

3. N Khi thời gian chạy của một chương trình là **tuyến**

tính, nói chung đây trường hợp mà một số lượng nhỏ các xử lý được làm cho mỗi phần tử dữ liệu nhập. Khi N là một triệu thì thời gian chạy cũng cỡ như vậy. Khi N được nhân gấp đôi thì thời gian chạy cũng được nhân gấp đôi. Đây là tình huống tối ưu cho một thuật toán mà phải xử lý N dữ liệu nhập (hay sản sinh ra N dữ liệu xuất).

3. NlogN Đây là thời gian chạy tăng dần lên cho các thuật toán mà giải một bài toán bằng cách tách nó thành các bài toán con nhỏ hơn, kế đến giải quyết chúng một cách độc lập và sau đó tổ hợp các lời giải. Bởi vì thiếu một tính từ tốt hơn (có lẽ là "tuyến tính logarit"?), chúng ta nói rằng thời gian chạy của thuật toán như thế là "NlogN". Khi N là một triệu, NlogN có lẽ khoảng hai mươi triệu. Khi N được nhân gấp đôi, thời gian chạy bị nhân lên nhiều hơn gấp đôi (nhưng không nhiều lắm).

4. N^2 Khi thời gian chạy của một thuật toán là **bậc hai**, trường hợp này chỉ có ý nghĩa thực tế cho các bài toán tương đối nhỏ. Thời gian bình phương thường tăng dần lên trong các thuật toán mà xử lý tất cả các cặp phần tử dữ liệu (có thể là hai vòng lặp lồng nhau). Khi N là một ngàn thì thời gian chạy là một triệu. Khi N được nhân đôi thì thời gian chạy tăng lên gấp bốn lần.

5. N^3 Tương tự, một thuật toán mà xử lý các bộ ba của các phần tử dữ liệu (có lẽ là ba vòng lặp lồng nhau) có thời gian chạy bậc ba và cũng chỉ có ý nghĩa thực tế trong các bài toán nhỏ. Khi N là một trăm thì thời gian chạy là một triệu. Khi N được nhân đôi, thời gian chạy tăng lên gấp tám lần.

6. 2^N Một số ít thuật toán có thời gian chạy lũy thừa lại thích hợp trong một số trường hợp thực tế, mặc dù các thuật toán như thế là "sự ép buộc thô bạo" để giải các bài toán. Khi N là hai mươi thì thời gian chạy là một triệu. Khi N gấp đôi thì thời gian chạy được nâng lên lũy thừa hai !

Thời gian chạy của một chương trình cụ thể đôi khi là một hằng số nhân với các số hạng nói trên ("số hạng dẫn đầu") cộng thêm một số hạng nhỏ hơn. Giá trị của hằng số và các số hạng phụ thuộc vào kết quả của sự phân tích và các chi tiết cài đặt. Hệ số của số hạng dẫn đầu liên quan tới số chỉ thị bên trong vòng lặp: ở một tầng tùy ý của thiết kế thuật toán thì phải cẩn thận giới hạn số chỉ thị như thế. Với N lớn thì các số hạng dẫn đầu đóng vai trò chủ chốt; với N nhỏ thì các số hạng cũng đóng góp vào và sự so sánh các thuật toán sẽ khó khăn hơn. Trong hầu hết các trường hợp, chúng ta sẽ gặp các chương trình có thời gian chạy là "tuyến tính", "NlogN", "bậc ba", ... với hiểu ngầm là các phân tích hay nghiên cứu thực tế phải được làm trong trường hợp mà tính hiệu quả là rất quan trọng.

$\lg N$	$\lg^2 N$	$\sqrt{N}$	N	$N \lg N$	$N \lg^2 N$	$N^{3/2}$	N^2
3	9	3	10	30	90	30	100
6	36	10	100	600	3,600	1,000	10,000
9	81	31	1,000	9,000	81,000	31,000	1,000,000
13	169	100	10,000	130,000	1,690,000	1,000,000	100,000,000
16	256	316	100,000	1,600,000	25,600,000	31,600,000	*ten billion*
19	361	1,000	1,000,000	19,000,000	361,000,000	*one billion*	*one trillion*

Hình 6.1 *Các giá trị xấp xỉ tương đối của các hàm*

Ngoài những hàm vừa nói trên cũng có một số hàm khác, ví dụ như một thuật toán với N^2 phần tử dữ liệu nhập mà có thời gian chạy là bậc ba theo N thì sẽ được phân lớp như một thuật toán $N^{3/2}$. Một số thuật toán có hai giai đoạn phân tách thành các bài toán con và có thời gian chạy xấp xỉ với $N\log^2 N$. Cả hai hàm này xem như gần với NlogN nhiều hơn gần với N^2 khi N lớn.

Cũng cần nên chú ý thêm về hàm "log", cơ số logarit chỉ làm thay đổi các đại lượng bởi một hệ số hằng. Bởi vì chúng ta thường sử dụng các kết quả phân tích sai khác một hằng số, nên hầu như không quan tâm đến cơ số là bao nhiêu khi đề cập tới "logN", chúng ta sẽ chú thích rõ ràng về cơ số khi cần thiết. Trong toán học, logarit tự nhiên (cơ số e=2.718281828...) được dùng thường

xuyên với viết tắt là: $\log_e N = \ln N$. Trong khoa học máy tính, logarit nhị phân (cơ số 2) được dùng thường xuyên với viết tắt là: $\log_2 N = \lg N$. Ví dụ, lgN khi làm tròn tới số nguyên gần nhất là số bit cần thiết để biểu diễn N dưới dạng nhị phân.

Hình 6.1 cho thấy kích thước tương đối của những hàm nầy: các giá trị xấp xỉ của lgN, $\lg^2 N$, sqrt(N), N, NlgN, $N\lg^2 N$, $N^{3/2}$, N^2 được cho với các giá trị khác nhau của N. Hàm bậc hai rõ ràng đóng vai trò chính yếu, đặc biệt với N lớn, và sự khác nhau giữa các hàm nhỏ hơn thì không đáng kể nếu N nhỏ. Ví dụ $N^{3/2}$ chỉ lớn hơn $N\lg^2 N$ với N rất lớn, nhưng trường hợp N nhỏ lại thường xảy ra trong thực tế. Bảng nầy không nhằm mục đích cho một so sánh giữa các hàm với mọi N-các số, các bảng, và các đồ thị liên quan tới từng thuật toán cụ thể có thể làm điều đó, nhưng bảng cũng cho ta một ý niệm ban đầu về việc phân loại các hàm.

ĐỘ PHỨC TẠP TÍNH TOÁN

Một cách tiếp cận để nghiên cứu tính năng của các thuật toán là nghiên cứu tính năng trong **trường hợp xấu nhất**, bỏ qua các hằng số, mục đích là xác định các phụ thuộc hàm số của thời gian chạy (hay một vài độ đo khác) vào số lần nhập (hay các đại lượng khác). Cách tiếp cận nầy rất hấp dẫn bởi vì nó cho phép ta chứng minh chặt chẽ các mệnh đề toán học về thời gian chạy của các chương trình: ví dụ, người ta có thể nói rằng thời gian chạy của thuật toán Mergesort (xem chương 11) thì tiệm cận tới NlogN.

Bước đầu tiên là làm chính xác về mặt toán học của khái niệm "tiệm cận tới", trong khi đối với phân tích một thuật toán thì ta tách sự phân tích khỏi một cài đặt cụ thể. Ý tưởng ở đây là bỏ qua các hệ số hằng trong phân tích: trong hầu hết các trường hợp nếu chúng ta muốn biết thời gian chạy của một thuật toán tiệm cận tới N hay tới logN, vấn đề không phải ở thuật được chạy trên một máy vi tính hay trên một siêu máy tính, cũng không phải ở bên trong các vòng lặp được cài đặt cẩn thận chỉ với một vài chỉ thị hay được cài đặt không tốt với rất nhiều chỉ thị. Về quan điểm toán học tất cả những tình huống vừa kể đều như nhau.

Một ký hiệu toán học cần được dùng được gọi là "ký hiệu O lớn" và viết là O được định nghĩa như sau:

Ký hiệu. Một hàm g(N) được nói là O(f(N)) nếu tồn tại các hằng số C_0 và N_0 sao cho g(N) nhỏ hơn C_0f(N) với mọi $N > N_0$.

Ký hiệu nầy giải phóng các người phân tích thuật toán khỏi sự khảo sát chi tiết về các đặc trưng của một máy cụ thể. Hơn nữa, điều khẳng định rằng thời gian chạy của thuật toán là O(f(N)) thì độc lập với dữ liệu nhập của thuật toán. Bởi vì chúng ta chú ý tới việc nghiên cứu thuật toán chứ không phải chú ý tới dữ liệu nhập hay cài đặt, ký hiệu O là một phương pháp hữu dụng để khẳng định các chận trên của thời gian chạy mà độc lập với cả các dữ liệu nhập lẫn các chi tiết cài đặt.

Ký hiệu O vô cùng có ích trong việc giúp đỡ các nhà phân tích thuật toán phân loại các thuật toán dựa vào tính năng và hướng dẫn các nhà thiết kế thuật toán trong việc tìm ra các thuật toán tốt nhất cho các bài toán quan trọng. Mục đích của việc nghiên cứu độ phức tạp của tính toán của một thuật toán là chỉ ra thời gian chạy của nó là O(f(N)) với một hàm f nào đó, và không có thời gian chạy là O(g(N)) với bất kỳ một hàm g(N) "nhỏ hơn" (nhỏ hơn theo nghĩa $\lim_{N \to \infty} [g(N)/f(N)] = 0$). Chúng ta cố gắng để cung cấp cả "cận trên" lẫn "cận dưới" của thời gian chạy trong trường hợp xấu nhất. Việc cung cấp các cận trên thường quan trọng trong việc đếm và đánh giá các tần số của chỉ thị (chúng ta sẽ thấy nhiều ví dụ trong các chương sau); việc cung cấp các chận dưới thì liên quan đến sự cẩn thận trong xây dựng một mô hình máy và xác định các thao tác cơ bản nào cần phải được thực hiện bởi thuật toán để giải một bài toán (chúng ta hiếm khi chạm tới vấn đề nầy). Khi các nghiên cứu tính toán cho thấy rằng cận trên của một thuật toán trùng với cận dưới của nó thì bảo đảm rằng không thể cố gắng thiết kế một thuật toán nhanh hơn về mặt cơ bản và chúng ta sẽ bắt đầu quan tâm đến sự cài đặt. Quan điểm nầy đã được công nhận là rất hữu ích đối với các nhà thiết kế thuật toán trong những năm gần đây.

Tuy nhiên, phải vô cùng cẩn thận khi áp dụng các kết quả từ ký hiệu O do bốn lý lẻ sau: trước tiên nó là một "chận trên" và đại

lượng trong câu hỏi có lẻ thấp hơn nhiều; thứ hai là dữ liệu nhập mà gây nên trường hợp xấu nhất có thể hiếm xảy ra trong thực tế; thứ ba là hằng số C_0 thì không được biết và không cần nhỏ; và thứ tư là hằng số N_0 không được biết và không cần nhỏ. Chúng ta sẽ khảo sát mỗi nhân tố này một cách chi tiết hơn.

Sự khẳng định rằng thời gian chạy của thuật toán là O(f(N)) không hàm ý rằng thời gian chạy đạt được giá trị đó, mà chỉ nói lên rằng các nhà phân tích đã chứng minh rằng thời gian chạy không bao giờ đạt tới đó. Thời gian chạy thông thường có thể luôn luôn nhỏ hơn nhiều. Ký hiệu tốt hơn đã được phát triển để giải quyết tình huống này là giả sử biết được rằng tồn tại một số dữ liệu nhập mà thời gian chạy là O(f(N)), nhưng có nhiều thuật toán lại rất khó xây dựng được dữ liệu nhập trong trường hợp xấu nhất.

Ngay cả nếu dữ liệu nhập trong trường hợp xấu nhất đã biết, cũng có thể xảy ra trường hợp mà đối với dữ liệu nhập thường gặp trong thực tế thì thời gian chạy thấp hơn nhiều. Có nhiều thuật toán vô cùng hữu ích mà lại có trường hợp xấu nhất rất dỡ tệ. ví dụ như thuật toán sắp xếp được dùng rộng rãi nhất là QuickSort có thời gian chạy là $O(N^2)$ nhưng có thể dàn xếp sao cho thời gian chạy đối với các dữ liệu nhập thường gặp trong thực tế là tiệm cận tới NlogN.

N	$\frac{1}{2}N\lg^2 N$	$\frac{1}{5}N\lg^2 N$	$N\lg^2 N$	$N^{3/2}$
10	22	45	90	30
100	900	1,800	3,600	1,000
1,000	20,250	40,500	81,000	31,000
10,000	422,500	845,000	1,690,000	1,000,000
100,000	6,400,000	12,800,000	25,600,000	31,600,000
1,000,000	90,250,000	180,500,000	361,000,000	1,000,000,000

Hình 6.2 *Sự ảnh hưởng của các hằng số khi so sánh các hàm*

Các hằng số C_0 và N_0 trong ký hiệu O thường che dấu các chi tiết cài đặt mà đóng vai trò quan trọng trong thực tế. Hiển nhiên rằng khi nói một thuật toán có thời gian chạy O(f(N)) thì sẽ không bàn gì được về thời gian chạy khi N nhỏ hơn N_0. Chúng ta thích một thuật toán dùng N^2 nano giây hơn thuật toán dùng logN thế kỷ, nhưng ký hiệu O thì không làm rõ được điều này, do đó ta thấy rằng hằng số C_0 đóng vai trò rất quan trọng. Hình 6.2 minh họa cho hai hàm thông thường cùng với nhiều giá trị cụ thể của các hằng số, và xét trong phạm vi 0<=N<=1000000. Hàm $N^{3/2}$ có thể bị tưởng lầm là lớn nhất trong bốn hàm thì lại là một trong những hàm nhỏ nhất đối với N nhỏ, và nó nhỏ hơn hàm Nlg^2N cho tới khi N vượt khỏi vài chục ngàn. Các chương trình mà thời gian chạy phụ thuộc vào các hàm như thế không thể được đánh giá tốt nếu không chú ý cẩn thận vào các hệ số hằng và các chi tiết cài đặt.

Khi phải quyết định chọn một từ thuật toán có thời gian chạy $O(N^2)$ và thuật toán có thời gian chạy O(N) thì không nên chỉ dựa vào độ phức tạp được biểu diễn bởi ký hiệu O. Với các cài đặt thực hành của các thuật toán trong quyển sách này, các chứng minh về độ phức tạp thì quá tổng quát và ký hiệu O lại không chính xác trong ứng dụng. Độ phức tạp tính toán phải được khảo sát ở bước đầu tiên trong quá trình phân tích một thuật toán để phát hiện nhiều chi tiết hơn về các tính chất của nó. Trong quyển sách này thì chúng ta quan tâm đến các bước sau đó gần với các cài đặt trong thực tế hơn.

PHÂN TÍCH TRƯỜNG HỢP TRUNG BÌNH

Một tiếp cận khác trong việc nghiên cứu tính năng của thuật toán là khảo sát **trường hợp trung bình**. Trong tình huống đơn giản nhất, chúng ta có thể đặc trưng chính xác các dữ liệu nhập của thuật toán: ví dụ một thuật toán sắp xếp có thể thao tác trên một mảng N số nguyên ngẫu nhiên, hay một thuật toán hình học có thể xử lý N điểm ngẫu nhiên trên mặt phẳng với các tọa độ nằm giữa 0 và 1. Kế đến là tính toán thời gian thực hiện trung bình của mỗi chỉ thị, và tính thời gian chạy trung bình của chương trình bằng cách nhân tần số sử dụng của mỗi chỉ thị với thời gian cần cho chỉ

thị đó, sau cùng cộng tất cả chúng với nhau. Tuy nhiên có ít nhất ba khó khăn trong cách tiếp cận nầy như thảo luận dưới đây.

Trước tiên là trên một số máy tính thì rất khó xác định chính xác số lượng thời gian đòi hỏi cho mỗi chỉ thị. Trường hợp xấu nhất thì đại lượng nầy bị thay đổi và một số lượng lớn các phân tích chi tiết cho một máy tính có thể không thích hợp đối với một máy tính khác. Đây chính là vấn đề mà các nghiên cứu về độ phức tạp tính toán cũng cần phải né tránh.

Thứ hai, chính việc phân tích trường hợp trung bình lại thường là đòi hỏi toán học quá khó. Do tính chất tự nhiên của toán học thì việc chứng minh các chận trên thì thường ít phức tạp hơn bởi vì không cần sự chính xác. Hiện nay chúng ta chưa biết được tính năng trong trường hợp trung bình của rất nhiều thuật toán.

Thứ ba (và chính là điều quan trọng nhất) trong việc phân tích trường hợp trung bình là mô hình dữ liệu nhập có thể không đặc trưng đầy đủ dữ liệu nhập mà chúng ta gặp trong thực tế. Ví dụ như làm thế nào để đặc trưng được dữ liệu nhập cho chương trình xử lý văn bảng tiếng Anh? Một tác giả đề nghị nên dùng các mô hình dữ liệu nhập chẳng hạn như "tập tin thứ tự ngẫu nhiên" cho thuật toán sắp xếp, hay "tập hợp điểm ngẫu nhiên" cho thuật toán hình học, đối với những mô hình như thế thì có thể đạt được các kết quả toán học mà tiên đoán được tính năng của các chương trình chạy trên các các ứng dụng thông thường. Chúng ta sẽ thấy một vài ví dụ (xem chương 9), mặc dù một số kết quả sẽ vượt ra ngoài phạm vi quyển sách nầy.

CÁC KẾT QUẢ TIỆM CẬN VÀ XẤP XỈ

Thông thường thì các kết quả của việc phân tích toán học thì không chính xác mà chỉ là một sự xấp xỉ: kết quả có thể là một biểu thức bao gồm một dãy các số hạng giảm. Như chúng ta đã quan tâm nhiều đến các chỉ thị trong vòng lặp của một chương trình, chúng ta cũng quan tâm nhiều tới **số hạng dẫn đầu** (số hạng lớn nhất) của một biểu thức toán học.

Ví dụ, giả sử rằng sau một quá trình phân tích toán học chúng

ta xác định được rằng một thuật toán có chỉ thị trong một vòng lặp được lặp NlgN lần về mặt trung bình và thời gian cần cho mỗi chỉ thị là a_0 micro giây, một bộ phận khác lặp N lần và thời gian cần cho mỗi chỉ thị là a_1 micro giây, và bộ phận khởi động chỉ thực hiện một lần với thời gian là a_2 micro giây. Khi đó thời gian chạy trung bình của chương trình (tính theo micro giây) là:

$a_0NlgN + a_1N + a_2$

Cũng có thể viết như sau:

$a_0NlgN + O(N)$.

Nếu chúng ta chỉ chú ý tới một trả lời xấp xỉ thì với N lớn chúng ta không cần tìm giá trị của $a_1$hay a_2 . Điều quan trọng là nếu khó xác định được các số hạng khác trong thời gian chạy chính xác thì ký hiệu O cho chúng ta một trả lời xấp xỉ khi N lớn mà không cần quan tâm tới các số hạng như thế.

Về mặt kỹ thuật, chúng ta không thể bỏ qua các các số hạng nhỏ, bởi vì ký hiệu O không nói gì về kích thước của hằng số C_0 : nó có thể rất lớn.

Thật ra, khi một hàm f(N) lớn xấp xỉ hơn hàm g(N) thì chúng ta dùng thuật ngữ "khoảng f(N)" với nghĩa là f(N)+O(g(N)). Trong những trường hợp như thế độc giả có thể xem rằng với N lớn thì đạilượng trong câu hỏi của chúng ta xấp xĩ với f(N). Chẳng hạn nếu như chúng ta biết rằng một đại lượng là N(N-1)/2 thì chúng ta nói nó "khoảng" $N^2/2$.

CÁC CÔNG THỨC TRUY HỒI CƠ SỞ

Như sẽ thấy trong các chương sau, phần lớn các thuật toán đều dựa trên việc phân rã đệ qui một bài toán lớn thành nhiều bài toán nhỏ hơn, rồi dùng các lời giải của các bài toán nhỏ để giải bài toán ban đầu. Thời gian chạy của các thuật toán như thế được xác định bởi kích thước và số lượng các bài toán con và giá phải trả của sự phân rã. Trong phần nầy chúng ta quan sát các phương pháp cơ sở để phân tích các thuật toán như thế và trình bày một vài công thức chuẩn thường được áp dụng trong việc phân tích nhiều thuật toán mà ta sẽ nghiên cứu.

Tính chất rất tự nhiên của một chương trình đệ qui là thời gian chạy cho dữ liệu nhập có kích thước N sẽ phụ thuộc vào thời gian chạy cho các dữ liệu nhập có kích thước nhỏ hơn: điều nầy được diễn dịch thành một công thức toán học được gọi là một **quan hệ truy hồi**. Các công thức như thế mô tả chính xác tính năng của các thuật toán tương ứng, do đó để có được thời gian chạy chúng ta phải giải các quan hệ truy hồi. Bây giờ chúng ta chú ý vào các công thức chứ không phải các thuật toán.

Công thức 1. Công thức nầy thường dùng cho các chương trình đệ qui mà có vòng lặp duyệt qua dữ liệu nhập để bỏ bớt một phần tử:

$C_N = C_{N-1} + N$, với $N >= 2$ và $C_1 = 1$

Lời giải: C_N khoảng $N^2/2$. Để giải một công thức truy hồi như trên, chúng ta lần lượt áp dụng chính công thức đó như sau:

$$\begin{aligned} C_N &= C_{N-1} + N \\ &= C_{N-2} + (N-1) + N \\ &= C_{N-3} + (N-2) + (N-1) + N \ldots \\ &= C_1 + 2 + \ldots + (N-2) + (N-1) + N \\ &= 1 + 2 + \ldots + (N-2) + (N-1) + N \\ &= N(N+1)/2. \end{aligned}$$

Công thức 2. Công thức truy hồi nầy dùng cho chương trình đệ qui mà chia dữ liệu nhập thành hai phần trong mỗi bước:

$C_N = C_{N/2} + 1$, với $N >= 2$ và $C_1 = 0$.

Lời giải: C_N khoảng lgN. Phương trình nầy vô nghĩa trừ khi N chẵn hay chúng ta giả sử rằng N/2 là một phép chia nguyên: bây giờ chúng ta giả sử rằng $N = 2^n$ để cho công thức luôn luôn có nghĩa. (Chú ý rằng n=lgN.) Chúng ta viết như sau:

$$C_{2^n} = C_{2^{n-1}} + 1 = C_{2^{n-2}} + 2 = C_{2^{n-3}} + 3 = \ldots = n$$

Công thức chính xác cho N tổng quát thì phụ thuộc vào biểu diễn nhị phân của N, nói chung C_N khoảng lgN với mọi N.

Công thức 3. Công thức nầy dùng cho chương trình đệ qui mà chia đôi dữ liệu nhập nhưng có thể kiểm tra mỗi phần tử của dữ liệu nhập.

$C_N = C_{N/2} + N$, với $N>=2$ và $C_1=0$.

Lời giải: C_N khoảng 2N. Tương tự trên, công thức nầy chính là tổng N + N/2 + N/4 + N/8 + ... (dĩ nhiên điều nầy chỉ chính xác khi N là một lũy thừa của 2). Nếu dãy là vô hạn thì đây là một chuỗi hình học đơn giản mà được ước lượng chính xác là 2N. Với trường hợp N tổng quát, lời giải chính xác phụ thuộc vào biểu diễn nhị phân của N.

Công thức 4. Công thức nầy dùng cho chương trình đệ qui mà duyệt tuyến tính xuyên qua dữ liệu nhập, trước khi, trong khi, hay sau khi dữ liệu nhập được chia đôi:

$C_N = 2C_{N/2} + N$, với $N>=2$ và $C_1=0$.

Lời giải: C_N khoảng NlgN. Công thức nầy áp dụng cho nhiều thuật toán theo phương pháp "chia để trị"

$$C_{2^n} = 2C_{2^{n-1}} + 2^n$$

$$\frac{C_{2^n}}{2^n} = \frac{C_{2^{n-1}}}{2^{n-1}} + 1$$

$$= \frac{C_{2^{n-2}}}{2^{n-2}} + 1 + 1$$

$$= \ldots$$

Lời giải cho công thức nầy rất giống như trong công thức 2, nhưng hãy chia hai vế của công thức cho 2^n trong bước thứ hai.

Công thức 5. Công thức nầy dùng cho chương trình đệ qui mà tách dữ liệu nhập thành hai phần ở mỗi bước giống như chương trình vẽ thước trong chương 5 của chúng ta.

$C_N = 2C_{N/2} + 1$, với $N>=2$ và $C_1=0$.

Lời giải: C_N khoảng 2N. Chứng minh giống như công thức 4.

Các biến dạng của những công thức nầy, chẳng hạn như điều

kiện khởi đầu khác nhau hay các số hạng thêm vào khác nhau một ít, có thể ước lượng bằng cách dùng cùng một kỹ thuật như trên, mặc dù vậy, độc giả cũng nên chú ý rằng một quan hệ truy hồi dường như tương tự với một quan hệ đã biết thì đôi khi lại khó giải hơn rất nhiều. (Có nhiều kỹ thuật tổng quát nâng cao để giải các phương trình như thế bằng phương pháp toán học). Chúng ta sẽ gặp một vài công thức truy hồi phức tạp hơn nhiều trong các chương tới, nhưng chúng ta chỉ thảo luận về lời giải của chúng khi chúng xuất xuất hiện.

BÀN LUẬN THÊM

Nhiều thuật toán trong quyển sách nầy có sự phân tích toán học chi tiết và các nghiên cứu về tính năng quá phức tạp đến nỗi không thể thảo luận ở đây. Thật ra nhờ vào các nghiên cứu cơ sở như thế mà chúng ta có thể giới thiệu nhiều thuật toán.

Không phải tất cả các thuật toán đều đáng giá để phân tích chi tiết như thế, thật ra trong suốt quá trình thiết kế, chỉ cần làm việc với các tính năng xấp xỉ để định hướng cho thiết kế mà không cần sa vào chi tiết. Khi thiết kế trở nên rõ ràng hơn thì sẽ phải phân tích, và lúc đó nhiều công cụ toán học tinh xảo được áp dụng. Thường thì quá trình thiết kế mà nghiên cứu độ phức tạp quá chi tiết có thể dẫn tới các thuật toán "lý thuyết" xa rời các ứng dụng cụ thể. Một sai lầm thường mắc phải là tin rằng sự phân tích bắt đầu từ các nghiên cứu độ phức tạp sẽ dẫn trực tiếp tới các thuật toán hiệu quả trong thực tế. Mặt khác, độ phức tạp tính toán thì lại là một công cụ hữu dụng cho đề nghị khởi đầu trong trong thiết kế để chọn ra các phương pháp mới.

Không nên dùng một thuật toán khi không biết về tính năng của nó: các tiếp cận trong chương nầy sẽ cho ta một khái niệm mở đầu về tính năng của thuật toán rất hữu ích để theo dõi những chương sau. Trong chương kế, chúng ta sẽ thảo luận các nhân tố khác đóng vai trò quan trọng trong việc chọn một thuật toán.

BÀI TẬP

1. Giả sử biết rằng thời gian chạy của một thuật toán là O(NlogN) và thời gian chạy của một thuật toán khác là $O(N^3)$. Điều này nói lên điều gì về tính năng tương đối giữa hai thuật toán này?

2. Giả sử biết rằng thời gian chạy của một thuật toán luôn luôn khoảng NlogN và thời gian chạy của một thuật toán khác là $O(N^3)$. Điều này nói lên điều gì về tính năng tương đối giữa hai thuật toán này?

3. Giả sử biết rằng thời gian chạy của một thuật toán luôn luôn khoảng NlogN và thời gian chạy của một thuật toán khác luôn luôn khoảng $O(N^3)$. Điều này nói lên điều gì về tính năng tương đối giữa hai thuật toán này ?

4. Cho biết sự khác nhau giữa O(1) và O(2).

5. Giải công thức truy hồi sau đây

   $C_N = C_{N/2} + N^2$, với $N=2$ và $C_1=0$

   khi N là một lũy thừa của 2.

6. Với giá trị nào của N thì $10N\lg N > 2N^2$?

7. Hãy viết một chương trình để tính chính xác giá trị của C_N trong công thức 2 như thảo luận trong chương 5. Hãy so sánh kết quả với lgN.

8. Chứng minh rằng lời giải chính xác của công thức 2 là lgN+O(1).

9. Hãy viết một chương trình đệ qui để tính số nguyên lớn nhất nhỏ hơn $\log_2 N$. (Hướng dẫn: với N=1, giá trị của hàm này cho N div 2 thì lớn hơn 1 so với giá trị của hàm này cho N.)

10. Hãy viết một chương trình dùng phép lặp cho vấn đề trong bài tập trước. Kế đó viết một chương trình mà tính toán bằng cách dùng thư viện hàm chuẩn của Pascal. Nếu có thể hãy so sánh tính năng của ba chương trình này trên máy tính của bạn.

7
CÀI ĐẶT THUẬT TOÁN

Như đã đề cập trong chương 1, trọng tâm của chúng ta trong quyển sách này là tập trung vào chính các thuật toán - khi bàn về mỗi thuật toán, ta coi hiệu quả của nó là một thành tố cốt yếu trong việc hoàn tất một tác vụ lớn hơn nào đó. Cả hai quan điểm này được biện minh vì các trường hợp như vậy xảy ra cho từng thuật toán một và do sự lưu ý kỹ lượng của chúng ta khi đi tìm một phương pháp hiệu quả để giải quyết một bài toán cũng thường hay dẫn tới một thuật toán ưu việt (và hiệu quả) hơn. Dĩ nhiên, mục tiêu hẹp này rất không thực tế, vì có nhiều yếu tố rất thực khác mà nó phải được xét đến khi giải một bài toán phức tạp với một máy tính. Trong chương này, ta sẽ bàn về các kết quả có liên quan đến việc tạo ra các thuật toán khá lý tưởng mà ta mô tả là hữu ích trong các ứng dụng thực tiễn.

Cuối cùng, các tính chất của thuật toán chỉ là một mặt của vấn đề - một máy tính có thể được dùng để giải một bài toán một cách hiệu quả chỉ khi bản thân bài toán được hiểu rõ. Việc khảo sát một cách cẩn thận các tính chất của các ứng dụng thì nằm ngoài phạm vi của cuốn sách này; dự định của chúng ta là cung cấp đủ các thông tin về các thuật toán cơ bản mà ta có thể tạo ra các quyết định thông minh về việc dùng chúng. Hầu hết các thuật toán ta sẽ xem xét đã được chứng minh là hữu ích đối với hàng loạt các ứng dụng. Phạm vi các thuật toán có thể dùng được để giải quyết các bài toán khác nhau tùy vào phạm vi các nhu cầu của các ứng dụng khác nhau. Không có thuật toán tìm kiếm "tốt nhất" (kiếm một ví dụ), nhưng một phương pháp có thể là hoàn toàn thích hợp khi ứng dụng vào một hệ thống giữ chỗ máy bay và một phương pháp khác có thể là thật hữu ích khi sử dụng ở các vòng lặp trong của một chương trình thám-mã.

Các thuật toán hiếm khi tồn tại một cách vu vơ, ngoại trừ có thể là ở trong đầu của các nhà thiết kế thuật toán lý thuyết, sáng

tạo ra các phương pháp mà không xem xét tới bất kỳ một cài đặt cuối cùng nào hết, hay các nhà lập trình hệ thống ứng dụng, muốn vận dụng các phương pháp đặc biệt để giải các bài toán mà nó được hiểu rõ nếu không có chúng. Việc thiết kế thuật toán thích hợp ám chỉ việc đặt một tư duy nào đó vào trong tầm ảnh hưởng tiềm tàng của các quyết định thiết kế lên trên các cài đặt, và việc lập trình các ứng dụng thích hợp ám chỉ việc đặt một suy nghĩ nào đó vào các phẩm chất về hiệu quả của các phương pháp cơ bản được sử dụng.

CHỌN MỘT THUẬT TOÁN

Như ta sẽ thấy trong các chương theo sau, thông thường có một số thuật toán dùng được để giải từng bài toán một, tất cả đều với các đặc tính khác nhau về hiệu quả, đi từ lời giải đơn giản, thô thiển (brute-force) (nhưng có thể không hiệu quả) cho tới một lời giải phức tạp, đã được tinh chế (well-tuned) (và thậm chí có thể là tối ưu). (Thường không đúng khi cho rằng một thuật toán càng hiệu quả, thì cài đặt phải càng phức tạp, vì một vài trong số các thuật toán tốt nhất của chúng ta thì khá đẹp và gọn, nhưng đối với các mục đích của việc thảo luận này, thì chúng ta hãy giả định là điều này đúng). Như đã tranh cãi ở trên, ta không thể quyết định dùng thuật toán nào đối với một bài toán mà không phân tích các yêu cầu của bài toán. Chương trình có được chạy thường hay không ? Các đặc trưng tổng quát của hệ thống máy tính sẽ được dùng là gì? Thuật toán có phải là một phần nhỏ của một ứng dụng lớn, hay ngược lại ?

Quy luật đầu tiên của việc cài đặt là trước tiên ta nên cài đặt thuật toán đơn giản nhất để giải một bài toán cho trước. Nếu thực tế bài toán cụ thể mà ta gặp hoá ra là đơn giản, thì thuật toán đơn giản có thể giải quyết bài toán và không cần làm thêm gì nữa; nếu một thuật toán tinh vi hơn được yêu cầu, thì bản cài đặt đơn giản sẽ cung cấp sự một kiểm chứng tính đúng đắn cho các trường hợp nhỏ và cung cấp một cái nền cho việc đánh giá các đặc trưng về hiệu quả.

Nếu một thuật toán sẽ được chạy chỉ một vài lần trên các trường hợp không quá lớn, thì chắc chắn được ưa chuộng hơn khi

bắt máy tính dùng thêm một ít thời gian để chạy một thuật toán ít hiệu quả hơn một chút thay vì bắt người lập trình bỏ ra một lượng thời gian đáng kể nữa để phát triển một thuật toán tinh vi. Dĩ nhiên, sẽ có một mối đe dọa mà ta gặp phải khi dùng chương trình nhiều lần hơn là so với hình dung ban đầu, như thế ta nên luôn luôn phải chuẩn bị để bắt đầu lại và cài đặt một thuật toán tốt hơn.

Nếu thuật toán sẽ được cài đặt như là một phần của một hệ thống lớn, thì bản cài đặt "thô thiển" sẽ cung cấp tính năng được yêu cầu theo một cách thức đáng tin cậy, và hiệu quả có thể được nâng lên theo một cách kiểm soát được bằng cách thay nó bởi một thuật toán tốt hơn sau này. Dĩ nhiên, ta chú ý không nên loại trừ các tùy chọn bằng cách cài đặt thuật toán theo một cách mà nó khó nâng cấp sau này, và ta nên chú ý thật kỹ vào các thuật toán mà nó tạo ra những khó khăn về tính hiệu quả khi nghiên cứu hiệu quả của hệ thống như là một tổng thể. Cũng vậy, trong những hệ thống lớn thường xảy ra trường hợp là các yêu cầu thiết kế của hệ thống bắt buộc ngay từ đầu là phải có thuật toán tốt nhất. Ví dụ, có lẽ một cấu trúc dữ liệu phổ biến của hệ thống là một dạng cụ thể của xâu liên kết hay cây, sao cho các thuật toán dựa trên cấu trúc cụ thể đó là được ưa thích hơn. Ngoài ra, ta nên chú ý vào các thuật toán sẽ được dùng khi tạo ra các quyết định rộng rãi cấp hệ thống như vậy, vì cuối cùng, thường tính hiệu quả của toàn hệ thống phụ thuộc vào tính hiệu quả của một vài thuật toán cơ sở, ví dụ như những thuật toán được bàn trong quyển sách này.

Nếu thuật toán sẽ chỉ chạy một vài lần, nhưng trên những bài toán rất lớn, thì ta muốn có một sự tin tưởng nào đó là nó sẽ sinh ra những kết xuất có ý nghĩa và muốn có một đánh giá nào đó về thời gian thực hiện lâu mau của nó. Một lần nữa, bản cài đặt đơn giản thường có thể là thật hữu ích trong việc cài đặt cho một cuộc chạy đường dài, kể cả việc phát triển công cụ để kiểm tra kết xuất.

Lỗi phổ biến nhất hay mắc phải trong khi chọn một thuật toán là bỏ qua các đặc tính về hiệu quả. Các thuật toán nhanh hơn thường phức tạp hơn, và các nhà cài đặt thường sẵn lòng chấp nhận một thuật toán chậm hơn để tránh phải đối phó với những phức tạp thêm vào. Nhưng một thuật toán nhanh hơn thường không phức tạp hơn nhiều lắm, và việc phải đối phó thêm với một

chút phức tạp nữa thì chỉ là một giá nhỏ phải trả để tránh phải đối phó với một thuật toán chậm. Người sử dụng của một số đáng ngạc nhiên các hệ thống máy tính đã đánh mất một lượng thời gian đáng kể khi phải đợi các thuật toán bậc hai hoàn tất trong khi ta có thể dùng các thuật toán bậc NlogN chỉ hơi phức tạp hơn chút ít nhưng lại chạy trong một khoảng thời gian ít hơn nhiều lần.

Lỗi phổ biến thứ hai hay mắc phải trong việc chọn một thuật toán là chú ý quá nhiều vào các đặc tính về hiệu quả. Một thuật toán bậc NlogN có thể chỉ hơi phức tạp hơn chút ít so với một thuật toán bậc hai đối với cùng một bài toán, nhưng một thuật toán NlogN tốt hơn có thể làm xuất hiện một sự gia tăng đáng kể về độ phức tạp (và có thể chỉ thực sự nhanh hơn khi gặp những giá trị N rất lớn). Cũng vậy, nhiều chương trình chỉ thực sự chạy một vài lần: thời gian cần để cài đặt và bắt lỗi một thuật toán đã được tối ưu có thể là lớn hơn rất nhiều so với thời gian cần chỉ để chạy một thuật toán chậm hơn chút ít.

PHÂN TÍCH THEO KINH NGHIỆM

Như đã đề cập trong chương 6, thật không may thường xảy ra trường hợp là việc phân tích toán học có thể soi rọi rất ít ánh sáng trên việc một thuật toán đã cho có thể được hình dung sẽ thực hiện trong một trường hợp cho trước tốt như thế nào. Trong các trường hợp như vậy, chúng ta cần dựa trên việc phân tích theo kinh nghiệm, trong đó chúng ta cài đặt một cách cẩn thận một thuật toán và giám sát hiệu quả của nó trên dữ liệu nhập "đặc trưng". Thực ra, điều này nên được thực hiện ngay cả khi ta có thể dùng được các kết quả toán học trọn vẹn, để kiểm tra tính hợp lệ của nó.

Cho trước hai thuật toán để giải cùng một bài toán, không có một sự bí hiểm nào trong phương pháp : chạy cả hai thuật toán này để xem cái nào chạy lâu hơn ! Điều này có vẻ là quá hiển nhiên phải lưu ý đến, nhưng đó có thể là sự bỏ sót phổ biến nhất trong việc nghiên cứu các phương pháp theo so sánh. Sự kiện mà một thuật toán nhanh gấp mười lần một thuật toán khác thì rất khó thoát khỏi sự chú ý của một người nào đó khi mà anh ta đợi một cái làm xong trong ba giây và phải đợi một cái khác làm xong

trong ba mươi giây, nhưng lại rất dễ xem nó như một thành tố hao phí nhỏ không đổi trong một phép phân tích toán học.

Tuy nhiên, cũng dễ mắc lỗi khi so sánh các cài đặt, đặc biệt nếu các máy, trình biên dịch hay các hệ thống khác nhau được đề cập đến, hay nếu các chương trình rất lớn với các dữ kiện nhập ít đặc trưng đang được so sánh với nhau. Thực vậy, một thành tố mà nó dẫn tới sự phát triển của việc phân tích toán học các thuật toán đã trở thành một xu hướng dựa trên các "benchmarks" mà hiệu quả của nó có thể được hiểu tốt hơn qua việc phân tích cẩn thận.

Mối nguy hiểm chính khi so sánh các chương trình theo kinh nghiệm là một bản cài đặt có thể được "tối ưu" hơn so với bản còn lại. Tác giả của một thuật toán mới được đề nghị có thể phải chú ý rất nhiều về mọi khía cạnh trong việc cài đặt của nó, và không chú ý vào các chi tiết của việc cài đặt một thuật toán cổ điển cạnh tranh với nó. Để tin tưởng vào độ chính xác của một nghiên cứu theo kinh nghiệm việc so sánh các thuật toán, ta phải bảo đảm là chú ý đồng đều đến các cài đặt. May mắn thay, thường xảy ra trường hợp: nhiều thuật toán ưu việt được phát sinh từ các sửa đổi tương đối nhỏ trên các thuật toán khác đối với cùng một bài toán, và các nghiên cứu so sánh thực sự là hợp lệ.

Một trường hợp đặc biệt quan trọng xảy ra khi một thuật toán được so sánh với một phiên bản khác của chính nó, hay những cách tiếp cận cài đặt khác nhau sẽ được so sánh với nhau. Một cách tuyệt vời để kiểm tra tính hiệu lực của một ý tưởng sửa đổi hay cài đặt cụ thể là chạy cả hai phiên bản trên cùng một dữ liệu nhập "cụ thể", sau đó chú ý đến cái chạy nhanh hơn. Một lần nữa, điều này hầu như có vẻ là quá hiển nhiên để ta phải lưu tâm đến, nhưng một số đáng ngạc nhiên các nhà nghiên cứu có liên quan trong lúc thiết kế thuật toán lại không bao giờ cài đặt các thiết kế của họ, như vậy làm cho người sử dụng phải đề phòng !

Như đã phác họa ở trên và ở đầu của chương 6, cách nhìn được dùng ở đây là tất cả việc thiết kế, cài đặt, phân tích toán học, và phân tích theo kinh nghiệm, đều đóng góp các phương pháp quan trọng cho việc phát triển các cài đặt tốt cho các thuật toán tốt. Chúng ta muốn dùng bất kỳ một công cụ nào có thể dùng được để

nhận được các thông tin về các tính chất của chương trình của mình, sau đó sửa đổi hay phát triển các chương trình mới trên cơ sở của các thông tin đó. Mặt khác, ta không luôn luôn được biện minh trong việc tạo ra một lượng lớn các thay đổi nhỏ với hy vọng cải tiến hiệu quả được chút ít. Tiếp theo, ta sẽ bàn luận về vấn đề này chi tiết hơn.

TỐI ƯU CHƯƠNG TRÌNH

Tiến trình tổng quát của việc tạo ra các sửa đổi ngày càng tiến bộ hơn cho một chương trình để sinh ra một phiên bản khác chạy nhanh hơn thì được gọi là tối ưu chương trình. Đây là một cái tên giả do có lẽ chúng ta không thể thấy được một cài đặt "tốt nhất" - chúng ta không thể tối ưu một chương trình, nhưng chúng ta hy vọng có thể cải tiến nó. Thông thường, việc tối ưu chương trình có liên quan đến các kỹ thuật tự động được áp dụng như là một thành phần của tiến trình biên dịch để cải tiến hiệu quả của chương trình đã được dịch. Ở đây chung ta dùng thuật ngữ này để liên hệ đến các sự cải tiến đặc - tả - thuật - toán (algorithm-specific). Dĩ nhiên, tiến trình cũng khá phụ thuộc vào môi trường lập trình và máy được sử dụng, như vậy ta sẽ chỉ xem xét các vấn đề khái quát ở đây, chứ không phải các kỹ thuật cụ thể.

Kiểu hoạt động này được biện minh chỉ khi ta bảo đảm là chương trình sẽ được dùng nhiều lần hay với một lượng dữ liệu nhập lớn và nếu thí nghiệm chứng minh là nỗ lực được đặt vào trong việc cải tiến bản cài đặt sẽ được tưởng thưởng bởi hiệu quả tốt hơn. Cách tốt nhất để cải tiến hiệu quả của một thuật toán là qua một tiến trình biến đổi dần dần chương trình thành các cài đặt ngày càng tốt hơn. Ví dụ khử-đệ-quy trong chương 5 là một thí dụ về một tiến trình như vậy, mặc dù việc cải tiến tính hiệu quả không phải là mục tiêu của chúng ta trong trường hợp đó.

Bước đầu tiên trong việc cài đặt một thuật toán là phát triển một bản chạy được của thuật toán trong dạng đơn giản nhất của nó. Điều này cung cấp một cái nền cho việc tinh chế và cải tiến, và như đã đề cập ở trên, thường là tất cả những gì được cần đến. Bất kỳ một kết quả toán học nào dùng được thì nên được kiểm chứng dựa trên bản cài đặt: ví dụ, nếu việc phân tích có vẻ như cho thấy

thời gian chạy là O(logN) nhưng thời gian chạy thực sự lại chỉ tính theo giây, thì có một cái gì đó sai lầm, hoặc là do cài đặt hay do phân tích, và cả hai nên được nghiên cứu cẩn thận hơn.

Bước kế tiếp là định danh "vòng lặp trong" và thử cực tiểu hoá số lệnh được đề cập đến. Có lẽ cách dễ nhất để tìm vòng lặp trong là chạy chương trình và sau đó kiểm tra xem những lệnh nào là được thực hiện thường nhất. Thông thường, đây là một dấu hiệu đặc biệt tốt để cho ta biết chương trình nên được cải tiến ở đâu. Mỗi lệnh của vòng lặp trong nên được xem xét kỹ lưỡng. Nó có thực sự cần thiết hay không ? Có cách nào hiệu quả hơn để đạt được cùng một nhiệm vụ hay không ? Ví dụ, thông thường ta phải trả giá để loại bỏ các lệnh gọi thủ tục từ vòng lặp trong. Có một số kỹ thuật "tự động" khác để làm điều này, nhiều cái trong số đó được cài đặt trong các trình biên dịch chuẩn. Cuối cùng, hiệu quả tốt nhất có được bằng cách chuyển vòng lặp trong thành ngôn ngữ máy hay hợp ngữ, nhưng thông thường đây là phương sách cuối cùng.

Không phải tất cả các "cải tiến" thực sự làm gia tăng hiệu quả, vì vậy việc kiểm tra phạm vi tiết kiệm nhận được ở mỗi bước là vô cùng quan trọng. Hơn nữa, khi cài đặt trở nên ngày càng được tinh chế hơn, thì khôn ngoan là nên xem xét lại khi nào thì những lưu ý cẩn thận như vậy về các chi tiết của chương trình là được biện minh. Trong quá khứ, thời gian chạy máy là quá đắt mà việc chi phí cho thời gian người lập trình để tiết kiệm các chu kỳ tính toán đã hầu như luôn luôn được biện minh, nhưng trong những năm gần đây tình thế đã thay đổi.

Ví dụ, xét thuật toán duyệt cây tiền-thứ-tự (preorder tree traversal) đã bàn trong chương 5. Thực sự, việc khử đệ quy là bước đầu tiên trong việc "tối ưu" thuật toán này, vì nó tập trung vào vòng lặp trong. Bản không đệ quy thì thực sự là chậm hơn bản đệ quy trên nhiều hệ thống (người đọc có thể kiểm tra điều này) do vòng lặp trong dài hơn và gồm bốn (mặc dù không đệ quy) lệnh gọi thủ tục (pop, push, push và stackempty) thay vì hai. Nếu các lệnh gọi tới các thủ tục của ngăn xếp được thay bằng đoạn chương trình để truy xuất trực tiếp ngăn xếp (bằng cách dùng một cài đặt mảng), chương trình này có thể là nhanh hơn đáng kể bản đệ quy.

(Một trong các thao tác push của thuật toán là vụn vặt, như vậy chương trình lặp-trong-vòng-lặp có thể là cơ sở cho một phiên bản được tối ưu hoá). Sau đó rõ ràng là vòng lặp trong liên quan đến việc tăng con trỏ ngăn xếp, chứa một con trỏ ($t^{\uparrow}.r$) trong mảng ngăn xếp, đặt lại con trỏ t (thành $t^{\uparrow}.l$), và so sánh nó với z. Trên nhiều máy, điều này có thể được cài đặt trong bốn lệnh của ngôn ngữ máy, mặc dù một trình biên dịch đặc trưng có thể sinh ra gấp hai lần số lệnh hay nhiều hơn. Chương trình này có thể được tạo để chạy nhanh hơn bốn hay năm lần bản cài đặt đệ quy đơn giản mà không phải làm quá nhiều việc.

Hiển nhiên, các vấn đề đang bàn luận ở đây là rất phụ thuộc vào máy và hệ thống. Ta không thể tham dự vào một thử nghiệm nghiêm túc để tăng tốc một chương trình mà không có một kiến thức chi tiết về hệ điều hành và môi trường lập trình. Bản tối ưu của một chương trình có thể trở nên khá yếu và khó thay đổi, và một trình biên dịch mới hay một hệ điều hành mới (chứ không đề cập đến một máy tính mới) có thể làm hư hoàn toàn một bản cài đặt đã được tối ưu hoá cẩn thận. Ngoài ra, chúng ta tập trung vào tính hiệu quả trong các cài đặt của mình bằng cách chú ý vào vòng lặp trong ở một cấp độ cao và bằng cách bảo đảm là tổng phí từ thuật toán là được cực tiểu hoá. Các chương trình trong quyển sách này được mã một cách chặt chẽ và phải chịu sự cải tiến khác nữa theo một phương thức đơn giản đối với bất kỳ một môi trường lập trình nào.

Việc cài đặt của một thuật toán là một chu trình phát triển của một chương trình, bắt lỗi nó và học các tính chất của nó, sau đó tinh chế lại bản cài đặt cho đến khi đạt tới một mức hiệu quả đáng mong muốn. Như đã bàn trong chương 6, việc phân tích toán học thường có thể trợ giúp trong tiến trình: trước tiên, để đề ra những thuật toán nào là các ứng viên hứa hẹn thực hiện tốt trong một bản cài đặt cẩn thận; thứ hai, để giúp kiểm chứng là bản cài đặt đang thực hiện như mong muốn. Trong một vài trường hợp, tiến trình này có thể dẫn tới việc phát hiện ra các sự kiện về bài toán mà nó đề xuất ra một thuật toán mới hay những cải tiến thật sự trong một bản cài đặt cũ.

THUẬT TOÁN VÀ HỆ THỐNG

Việc cài đặt của các thuật toán trong quyển sách này có thể được tìm thấy trong hàng loạt các bài toán lớn, các hệ điều hành, và các hệ thống ứng dụng. Dự định của chúng ta là mô tả các thuật toán và khuyến khích người đọc tập trung vào các tính chất động qua việc thí nghiệm với các cài đặt đã cho. Đối với một vài ứng dụng, các cài đặt có thể thật hữu ích đúng như đã cho, nhưng đối với các ứng dụng khác có thể cần làm nhiều công việc hơn.

Trước tiên, như đã đề cập trong chương 2, các chương trình trong quyển sách này chỉ dùng các đặc trưng cơ bản của Pascal, thay vì tận dụng các khả năng cao cấp hơn mà nó có thể dùng được trong Pascal và các môi trường lập trình khác. Mục đích của chúng ta là nghiên cứu các thuật toán, chứ không phải là lập trình hệ thống hay các đặc trưng cấp cao của ngôn ngữ lập trình. Hy vọng là các đặc trưng chủ yếu của các thuật toán được trình bày một cách tốt nhất qua các cài đặt đơn giản, trực tiếp trong một ngôn ngữ gần-như-phổ-dụng.

Kiểu thức lập trình chúng ta dùng hơi vắn tắt, với các tên biến ngắn và một vài chú thích, sao cho các cấu trúc điều khiển được nổi bật lên. "Sưu liệu" của các thuật toán là các văn bản đi kèm. Mong rằng người đọc mà có dùng đến các chương trình này trong các ứng dụng thực sự sẽ sửa sang lại chúng một chút, làm cho chúng thích nghi được với một ứng dụng cụ thể. Một kiểu thức lập trình "đối phó" khác được biện hộ trong việc xây dựng các hệ thống thực: các chương trình phải được cài đặt sao cho chúng có thể được thay đổi dễ dàng, được đọc và hiểu nhanh chóng bởi các người lập trình khác, và giao tiếp tốt với các phần khác của hệ thống.

Cụ thể, các cấu trúc dữ liệu được yêu cầu cho các ứng dụng thường chứa các thông tin khác ngoài những cái được dùng trong quyển sách này, mặc dù các thuật toán mà chúng ta xem xét là thích hợp cho các cấu trúc dữ liệu phức tạp hơn. Ví dụ, ta nói về việc tìm kiếm xuyên suốt các tập tin chứa các số nguyên hay các chuỗi ký tự ngắn, trong khi một ứng dụng đặc trưng sẽ yêu cầu xét những chuỗi ký tự dài mà nó là một phần của các mẩu tin lớn.

Nhưng các phương pháp cơ bản dùng được cho cả hai trường hợp là như nhau. Trong các trường hợp như vậy, ta sẽ bàn đến các đặc trưng hiển nhiên của từng thuật toán và việc chúng có liên quan như thế nào với các yêu cầu ứng dụng khác nhau.

Nhiều thành phần trên đây liên quan đến việc tăng tính hiệu quả của một thuật toán cụ thể cũng được áp dụng để tăng tính hiệu quả của một hệ thống lớn. Tuy nhiên, trên phạm vi lớn hơn này, một kỹ thuật để tăng tính hiệu quả của hệ thống có thể là thay thế một cài đặt đơn thể của một thuật toán bằng một cài đặt đơn thể khác. Một nguyên lý cơ bản của việc xây dựng các hệ thống lớn là những thay đổi như vậy là có thể được. Cụ thể, khi một hệ thống lớn phát triển trong thực tế, những tri thức chính xác hơn về các yêu cầu cụ thể cho các đơn thể cụ thể được tăng cường. Tri thức cụ thể hơn này khiến cho ta có thể chọn một cách cẩn thận hơn thuật toán tốt nhất để dùng nhằm thoả mãn các yêu cầu đó; sau đó ta có thể tập trung vào việc cải tiến hiệu quả của thuật toán đó, như đã mô tả ở trên. Chắc chắn đây là trường hợp khi mà một phần lớn của chương trình hệ thống chỉ được thi hành một vài lần (hay không thi hành lần nào hết) - quan tâm chủ yếu của người xây dựng hệ thống là tạo ra một tổng thể cô kết. Mặt khác, rất có thể là khi một hệ thống đi vào sử dụng, nhiều tài nguyên của chúng sẽ được dùng để giải quyết các bài toán cơ bản có kiểu đã được bàn trong cuốn sách này, sao cho nó thích hợp cho các nhà xây dựng hệ thống để hiểu về các thuật toán cơ sở mà ta bàn đến.

BÀI TẬP

1. Mất bao nhiêu lâu để đếm tới 100000 ? Đánh giá xem chương trình j:=0; for i:=1 to 100000 do j:=j+1 chạy mất bao lâu trên môi trường lập trình của bạn, sau đó chạy chương trình để kiểm tra đánh giá của bạn.

2. Trả lời câu hỏi trên đây dùng repeat và while.

3. Bằng cách chạy trên các gía trị nhỏ, đánh giá xem cài đặt của sàng Eratosthenes trong chương 3 chạy mất bao lâu với N=1000000 (nếu có đủ bộ nhớ để dùng).

4. "Tối ưu" cài đặt sàng Eratosthenes trong chương 3 để tìm số nguyên tố lớn nhất bạn có thể có trong 10 giây tính toán.

5. Hãy kiểm tra khẳng định trong bài học là việc khử đệ quy từ thuật toán duyệt cây tiền-thứ-tự của chương 5 (với các lệnh gọi thủ tục cho các thao tác ngăn xếp) khiến cho chương trình chạy chậm hơn.

6. Hãy kiểm tra khẳng định trong bài học là việc khử đệ quy từ thuật toán duyệt cây tiền-thứ-tự của chương 5 (và việc cài đặt các thao tác ngăn xếp "inline") khiến cho chương trình chạy nhanh hơn.

7. Xét chương trình hợp ngữ được sinh bởi trình biên dịch Pascal trong môi trường lập trình cục bộ của bạn cho thuật toán duyệt cây tiền-thứ-tự đệ quy của chương 5.

8. Thiết kế một thử nghiệm để kiểm tra xem cài đặt xâu liên kết hay mảng của một ngăn xếp đẩy xuống (pushdown stack) là hiệu quả hơn trong môi trường lập trình của bạn.

9. Phương pháp nào hiệu quả hơn, không đệ quy hay đệ quy, để vẽ một thước kẻ đã cho trong chương 5.

10. Có đúng bao nhiêu động tác push vào ngăn xếp phụ trợ được dùng bởi cài đặt không đệ quy đã cho trong chương 5 khi duyệt một cây đầy đủ gồm $2^n - 1$ nút theo tiền-thứ-tự.

8
CÁC PHƯƠNG PHÁP SẮP XẾP CƠ BẢN

Vì cuộc du ngoạn đầu tiên của chúng ta là đi vào lãnh vực của các thuật toán sắp xếp, ta sẽ nghiên cứu một vài phương pháp "cơ bản" mà nó thích hợp cho các tập tin nhỏ hay các tập tin với một cấu trúc dữ liệu đặc thù nào đó. Có nhiều lý do để nghiên cứu những thuật toán sắp xếp đơn giản này chi tiết hơn. Đầu tiên, chúng cung cấp một phương pháp tương đối không khó khăn để học danh từ chuyên môn và các cơ chế cơ bản cho các thuật toán sắp xếp sao cho ta có được một cái nền thích hợp để nghiên cứu những thuật toán tinh vi hơn. Thứ hai là, trong nhiều ứng dụng lớn của sắp xếp, tốt hơn nên dùng các phương pháp đơn giản này thay vì là các phương pháp đa năng mạnh hơn. Cuối cùng, một vài trong số các phương pháp đơn giản này mở rộng thành các phương pháp đa năng tốt hơn hay có thể được dùng để nâng cao tính hiệu quả của các phương pháp mạnh hơn.

Như đã đề cập ở trên, có nhiều ứng dụng sắp xếp trong đó một thuật toán tương đối đơn giản có thể là phương pháp được chọn. Các chương trình sắp xếp thường chỉ được dùng một lần (hay chỉ một vài lần). Nếu số mẩu tin được sắp là không quá lớn (khoảng ít hơn năm trăm phần tử), có thể sẽ hiệu quả hơn khi chỉ chạy một phương pháp đơn giản thay vì cài đặt và bắt lỗi một phương pháp phức tạp. Các phương pháp cơ bản luôn thích hợp cho các tập tin nhỏ (ít hơn 500 phần tử); không thích hợp khi mà một thuật toán tinh vi sẽ được hiệu chỉnh cho một tập tin nhỏ, trừ phi ta cần sắp xếp một lượng rất lớn các tập tin như vậy. Các kiểu tập tin khác mà nó tương đối dễ sắp là những cái mà nó đã hầu như được sắp (hay đã được sắp !) hay những tập tin mà nó chứa một lượng lớn các khoá bằng nhau. Các phương pháp đơn giản có thể làm tốt hơn

nhiều trên các tập tin có cấu trúc tốt như vậy so với các phương pháp đa năng.

Như thông lệ, các phương pháp cơ bản mà ta xem xét sẽ lấy khoảng N^2 bước để sắp N mẩu tin được xếp ngẫu nhiên. Nếu N tương đối nhỏ, thì điều này có thể không thành vấn đề, và nếu các mẩu tin không được xếp một cách ngẫu nhiên, một vài phương pháp có thể chạy nhanh hơn những phương pháp phức tạp khác nhiều. Tuy nhiên, phải nhấn mạnh là các phương pháp này không nên dùng cho các tập tin lớn, sắp ngẫu nhiên, với ngoại lệ đáng chú ý là Shellsort, mà nó thực sự là phương pháp sắp xếp được chọn lựa cho một lượng lớn các ứng dụng.

QUY LUẬT CỦA TRÒ CHƠI

Trước khi xem xét một vài thuật toán cụ thể, cũng cần bàn về một vài thuật ngữ chuyên môn thông thường và các giả định cơ sở cho các thuật toán sắp xếp. Ta sẽ xem xét các phương pháp để sắp các tập tin gồm các mẩu tin có chứa khoá (key). Các khoá, mà nó chỉ là một phần của các mẩu tin (thường là một phần nhỏ), được dùng để điều khiển việc sắp xếp. Mục đích của phương pháp sắp xếp là tổ chức lại các mẩu tin sao cho các khoá của chúng được sắp thứ tự tương ứng với một quy luật sắp xếp đã được định nghĩa rõ ràng (thường là thứ tự số hay chữ).

Nếu tập tin được sắp nằm khít trong bộ nhớ (hay, trong ngữ cảnh của chúng ta, là nó nằm vừa trong một mảng của Pascal), thì phương pháp sắp được gọi là sắp xếp trong (internal sorting). Việc sắp các tập tin trên băng từ hay đĩa được gọi là sắp xếp ngoài (external sorting). Sự khác nhau chính giữa hai phương pháp là bất kỳ một mẩu tin nào cũng có thể dễ dàng được truy xuất trong một sự sắp xếp trong, trong khi một sắp xếp ngoài phải truy xuất các mẩu tin một cách tuần tự, hay ít nhất là theo các khối lớn. Ta sẽ xem một vài cách sắp xếp ngoài trong chương 13, nhưng hầu hết các thuật toán mà ta sẽ xem xét là sắp xếp trong.

Thông thường, thông số về hiệu quả chính mà ta quan tâm là thời gian chạy của các thuật toán sắp xếp. Bốn phương pháp đầu mà ta sẽ xem xét trong chương này cần một khoảng thời gian tỉ lệ

với N^2 để sắp N phần tử, trong khi các phương pháp cao cấp hơn có thể sắp N phần tử trong một khoảng thời gian tỉ lệ với NlogN. (Có thể chứng minh là không có thuật toán sắp nào có thể dùng ít hơn NlogN phép so sánh giữa các khoá). Sau khi xem xét các phương pháp đơn giản, ta sẽ xem một phương pháp cao cấp hơn mà nó có thể chạy trong khoảng thời gian tỉ lệ với $N^{3/2}$ hay ít hơn, và ta sẽ thấy là có những phương pháp mà nó dùng đến các tính chất số của các khoá để nhận được một thời gian chạy tổng cộng tỉ lệ với N.

Lượng bộ nhớ phụ trợ được dùng bởi một thuật toán sắp xếp là thành tố quan trọng thứ hai mà ta sẽ xem xét. Về cơ bản, các phương pháp chia thành ba loại: những phương pháp sắp tại chỗ và không dùng đến bộ nhớ phụ trợ ngoại trừ bộ nhớ cho ngăn xếp hay bảng; những phương pháp mà nó dùng một biểu diễn xâu liên kết và vì vậy dùng thêm N từ nhớ cho các con trỏ xâu; và các phương pháp mà nó cần có đủ bộ nhớ phụ trợ để chứa một bản sao khác của mảng sẽ được sắp.

Một đặc trưng của các phương pháp sắp xếp đôi khi quan trọng trong thực tiễn là tính ổn định (stability). Một phương pháp sắp được gọi là ổn định nếu nó bảo toàn thứ tự tương đối của các khoá bằng nhau trong tập tin. Ví dụ, nếu một danh sách lớp theo alphabet được sắp theo điểm, thì một phương pháp ổn định sẽ sinh ra một danh sách trong đó các sinh viên có cùng điểm thì vẫn nằm theo thứ tự alphabet, nhưng một phương pháp không ổn định có thể sẽ sinh ra một danh sách mà nó không còn một tàn tích gì về thứ tự alphabet ban đầu. Hầu hết các phương pháp đơn giản là ổn định, nhưng hầu hết các phương pháp phức tạp nổi tiếng thì không. Nếu tính ổn định là quan trọng, nó có thể bị bắt buộc bằng cách thêm vào một chỉ mục (index) nhỏ cho mỗi khoá trước khi sắp hay bằng cách nối dài khoá sắp ra theo một cách nào đó. Dễ làm cho tính ổn định được thừa nhận: con người thông thường phản ứng lại các tác động không ưng ý của tính bất ổn định bằng sự không tin tưởng. Thực ra, một vài phương pháp đạt được tính ổn định mà không dùng đến thời gian hay không gian phụ trợ.

Chương trình sau đây sắp xếp ba mẩu tin, dự định dùng để minh họa các quy ước thông thường mà ta sẽ sử dụng. Cụ thể,

chương trình chính là một chương trình kỳ dị để tập dượt, mà nó chỉ chạy với N=3; quan điểm là bất kỳ một chương trình sắp xếp nào cũng có thể được thay thế cho sort3 trong chương trình "điều khiển" này.

```
program threesort (input, output);
const maxN = 100;
var a: array [1..maxN] of integer; N,i: integer;
procedure sort3;
   var t:integer;
   begin
      if a[1] > a[2] then begin t := a[1]; a[1] := a[2]; a[2] := t end;
      if a[1] > a[3] then begin t := a[1]; a[1] := a[3]; a[3] := t end;
      if a[2] > a[3] then begin t := a[2]; a[2] := a[3]; a[3] := t end
   end;
begin readln(N);
for i:=1 to N do read(a[i]);
if N=3 then sort3;
for i:=1 to N do write(a[i]);
writeln
end.
```

Ba lệnh gán theo sau mỗi một if thực sự cài đặt một thao tác "hoán vị". Ta sẽ viết các đoạn mã cho những phép hoán vị như vậy thay vì dùng một lệnh gọi thủ tục vì chúng là cơ sở cho nhiều chương trình sắp xếp và thường rơi vào vòng lặp trong.

Với mục đích là tập trung vào các vấn đề thuật toán, ta sẽ làm việc với các thuật toán mà nó chỉ sắp các mảng số nguyên theo thứ tự số. Thường là đơn giản để làm cho các thuật toán như vậy trở nên thích nghi với một ứng dụng thực tế có các khoá lớn hay các mẩu tin lớn. Cơ bản, các chương trình sắp truy xuất các bản ghi theo một trong hai cách: hoặc là các khoá được truy xuất để so sánh, hoặc là toàn bộ các mẩu tin được truy xuất sẽ được di chuyển đi. Hầu hết các thuật toán ta sẽ nghiên cứu có thể được tính toán lại trên cơ sở hai thao tác này trên các mẩu tin tùy ý. Nếu các mẩu tin sẽ được sắp là lớn, thường sẽ là khôn ngoan khi tránh làm rối tung chúng lên bằng cách thực hiện một "phép sắp gián tiếp": ở đây chính các mẩu tin là không cần thiết được sắp lại, nhưng thay vào đó là một mảng các con trỏ (hay chỉ mục) được sắp xếp lại sao

cho con trỏ đầu tiên chỉ tới mẩu tin nhỏ nhất,... Các khoá có thể được giữ hoặc là với các mẩu tin (nếu chúng lớn) hoặc là với các con trỏ (nếu chúng nhỏ). Nếu cần, các mẩu tin sau đó có thể được tổ chức lại sau khi sắp xếp, như được mô tả trong phần sau của chương này.

Chương trình sort3 ở trên dùng một truy xuất thậm chí bị ràng buộc hơn với tập tin: đó là ba lệnh có dạng "so sánh hai mẩu tin và hoán vị chúng nếu cần để đặt mẩu tin có khoá nhỏ hơn lên đầu tiên". Các chương trình bị ràng buộc vào các lệnh như vậy là đáng chú ý vì chúng thích hợp cho cài đặt trong phần cứng. Ta sẽ nghiên cứu vấn đề này chi tiết hơn trong chương 40.

Bằng cách dùng những chương trình mà nó chỉ thao tác trên một mảng toàn cục, ta sẽ bỏ qua "bài toán nén" mà nó có thể gây phiền hà trong một vài môi trường lập trình. Mảng có nên đưa tới thủ tục sắp xếp như một tham biến hay không ? Có thể cùng một thủ tục sắp xếp được dùng để sắp các mảng số nguyên và các mảng số thực (và mảng gồm các mẩu tin phức tạp tùy ý) hay không ? Thậm chí với các giả định đơn giản của chúng ta, chúng ta phải dùng kế để tránh được sự thiếu sót các kích thước mảng động trong Pascal bằng cách khai báo trước một con số tối đa. Những mối quan tâm như vậy sẽ là dễ đối phó trong các môi trường lập trình tương lai hơn là trong các môi trường lập trình đã qua và hiện nay. Ví dụ, một vài ngôn ngữ lập trình hiện đại có các phương tiện được phát triển thật tốt để đóng gói các chương trình với nhau vào trong các hệ lớn. Tuy nhiên, những cơ chế như vậy thì không thực sự được yêu cầu đối với nhiều ứng dụng: các chương trình nhỏ mà nó hoạt động trực tiếp trên các mảng toàn cục có nhiều ứng dụng, và một vài hệ điều hành làm cho nó trở nên hoàn toàn dễ dàng để kết hợp các chương trình đơn giản lại với nhau, giống như cái ở trên, được dùng như một "bộ lọc" (filter) giữa ngõ nhập vào (input) và ngõ xuất ra (output) của nó. Hiển nhiên, các chú giải này áp dụng cho nhiều thuật toán khác mà ta sẽ xem xét, mặc dù các tác động đã được đề cập có lẽ đã được cảm nhận một cách sâu sắc nhất đối với các thuật toán sắp xếp.

Một số chương trình dùng một ít các biến toàn cục khác. Các khai báo mà nó không hiển nhiên thì sẽ được đưa vào trong đoạn

mã chương trình. Cũng vậy, ta sẽ đôi khi giả định là các biên của mảng đi tới 0 hay N+1, để chứa các khoá đặc biệt được dùng bởi một vài thuật toán. Ví dụ, thông thường ta sẽ dùng các chữ cái của bảng chữ cái thay vì dùng các số: các khoá này được kiểm soát theo một cách hiển nhiên, sử dụng các "hàm chuyển đổi" giữa số nguyên và ký tự là ord và chr của Pascal.

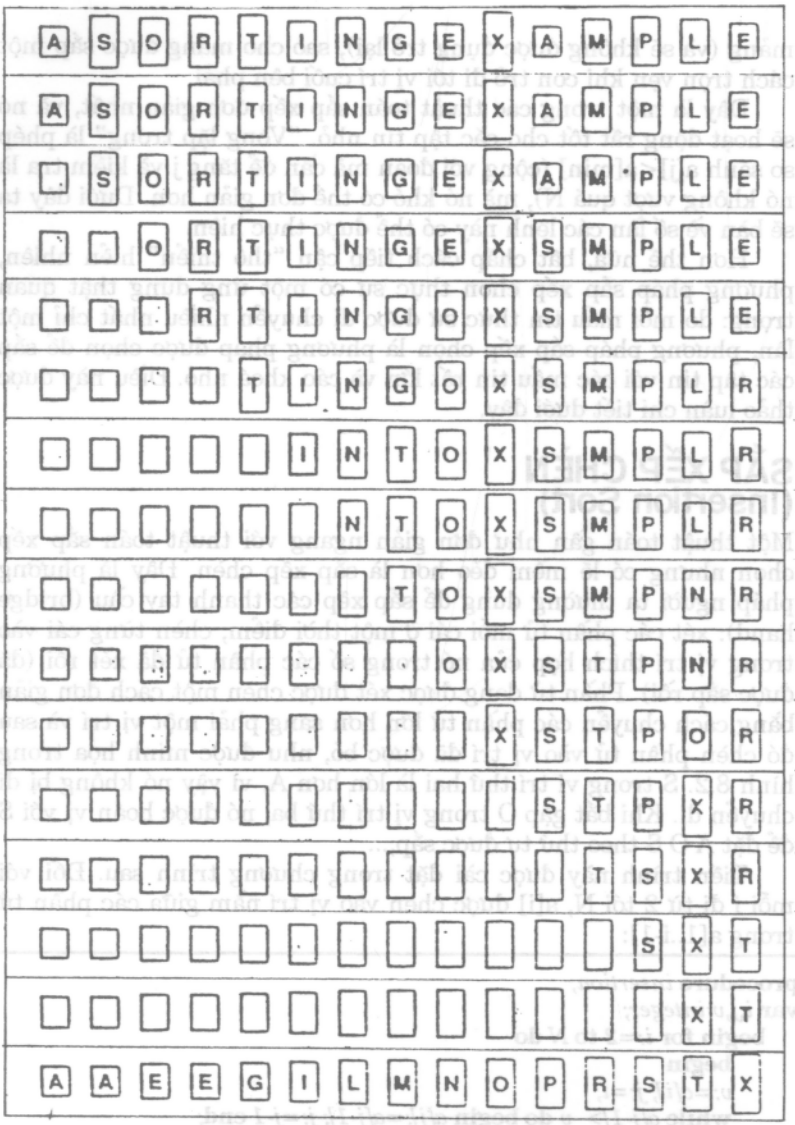
SẮP XẾP CHON (Selection Sort)

Một trong các thuật toán sắp xếp đơn giản nhất hoạt động như sau: đầu tiên tìm phần tử nhỏ nhất trong mảng và hoán vị nó với phần tử trong vị trí đầu tiên, sau đó tìm phần tử nhỏ nhất kế tiếp và hoán vị nó với phần tử trong vị trí thứ hai, và tiếp tục theo phương pháp này cho đến khi toàn bộ mảng đã được sắp. Phương pháp này được gọi là sắp xếp chọn vì nó làm việc bằng cách lặp lại việc "chọn" phần tử nhỏ nhất còn lại, như được minh hoạ trong hình 8.1. Bước đầu tiên không có ảnh hưởng gì do không có phần tử nào trong mảng nhỏ hơn A ở bên trái. Ở bước thứ hai, A thứ hai là phần tử nhỏ nhất còn lại, như thế nó được hoán vị với S trong vị trí thứ hai. Chữ E đầu tiên được hoán vị với O trong vị trí thứ ba ở bước thứ ba, sau đó chữ E thứ hai được hoán vị với R trong vị trí thứ tư ở bước thứ tư,...

Chương trình sau đây là một cài đặt đầy đủ của tiến trình này. Với mỗi i chạy từ 1 tới N-1, nó hoán vị phần tử nhỏ nhất trong a[i..N] với a[i]:

```
procedure selection;
  var i,j,min,t:integer;
  begin
    for i:=1 to N-1 do
      begin min:=i;
        for j:=i+1 to N do if a[j] < a[min] then min:=j;
        t:=a[min]; a[min]:=a[i]; a[i]:=t
      end;
  end;
```

Khi con trỏ i duyệt từ trái sang phải qua tập tin, các phần tử nằm bên trái của con trỏ là theo thứ tự kết quả của chúng trong



Hình 8.1 *Sắp xếp chọn*

mảng (và sẽ không được dụng trở lại), sao cho mảng được sắp một cách trọn vẹn khi con trỏ đi tới vị trí cuối bên phải.

Đây là một trong các thuật toán sắp xếp đơn giản nhất, và nó sẽ hoạt động rất tốt cho các tập tin nhỏ. "Vòng lặp trong" là phép so sánh a[j]<a[min] (cộng với đoạn mã cần để tăng j và kiểm tra là nó không vượt quá N), mà nó khó có thể đơn giản hơn. Dưới đây ta sẽ bàn về số lần các lệnh này có thể được thực hiện.

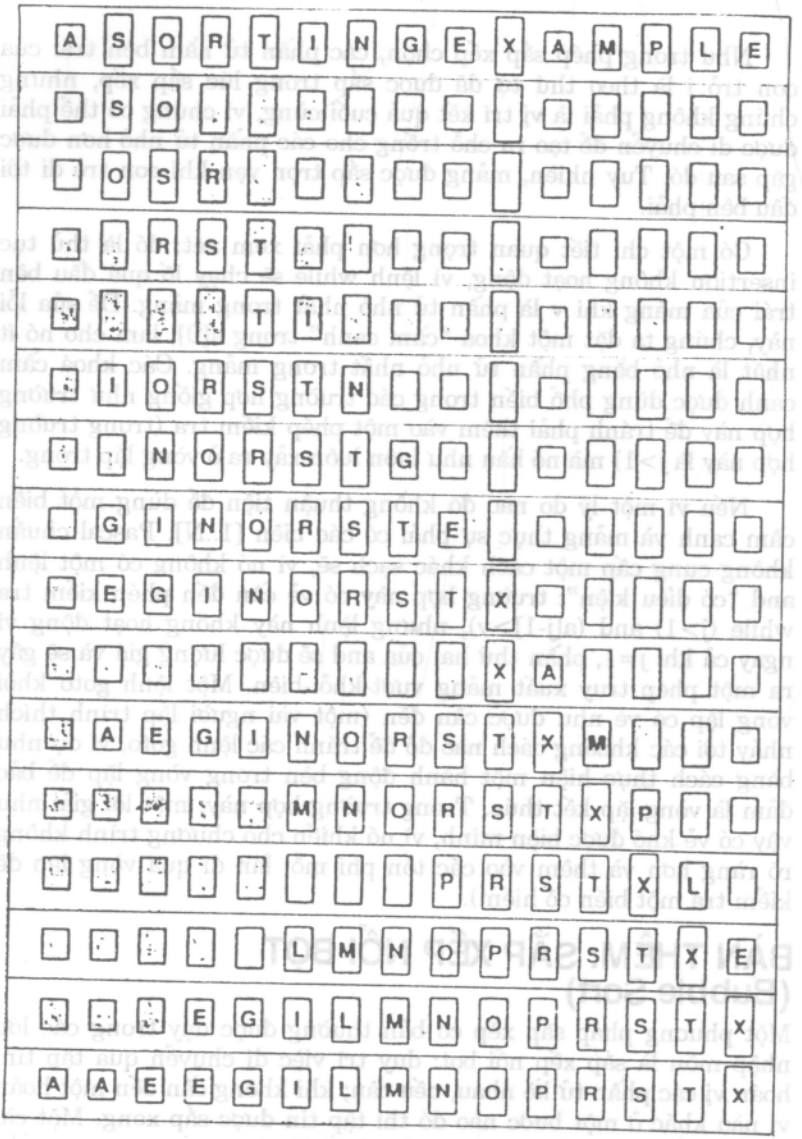
Hơn thế nữa, bất chấp cách tiếp cận "thô thiển" hiển nhiên, phương pháp sắp xếp chọn thực sự có một ứng dụng thật quan trọng: do mỗi mẩu tin thực sự được di chuyển nhiều nhất chỉ một lần, phương pháp sắp xếp chọn là phương pháp được chọn để sắp các tập tin với các mẩu tin rất lớn và các khoá nhỏ. Điều này được thảo luận chi tiết dưới đây.

SẮP XẾP CHÈN (Insertion Sort)

Một thuật toán gần như đơn giản ngang với thuật toán sắp xếp chọn nhưng có lẽ mềm dẻo hơn là sắp xếp chèn. Đây là phương pháp người ta thường dùng để sắp xếp các thanh tay cầu (bridge hand): xét các phần tử mỗi cái ở một thời điểm, chèn từng cái vào trong vị trí thích hợp của nó trong số các phần tử đã xét rồi (đã được sắp rồi). Phần tử đang được xét được chèn một cách đơn giản bằng cách chuyển các phần tử lớn hơn sang phải một vị trí và sau đó chèn phần tử vào vị trí đã được bỏ, như được minh họa trong hình 8.2. S trong vị trí thứ hai là lớn hơn A, vì vậy nó không bị di chuyển đi. Khi bắt gặp O trong vị trí thứ ba, nó được hoán vị với S để đặt A O S theo thứ tự được sắp,...

Tiến trình này được cài đặt trong chương trình sau. Đối với mỗi i đi từ 2 tới N, a[i] được chèn vào vị trí nằm giữa các phần tử trong a[1..i-1]:

```
procedure insertion;
var i,j,v:integer;
   begin for i:=2 to N do
      begin
      v:=a[i]; j:=i;
      while a[j-1]> v do begin a[j]:=a[j-1]; j:=j-1 end;
      a[j]:=v
      end
   end;
```



Hình 8.2 *Sắp xếp chèn*

Như trong phép sắp xếp chọn, các phần tử nằm bên trái của con trỏ i là theo thứ tự đã được sắp trong lúc sắp xếp, nhưng chúng không phải là vị trí kết quả cuối cùng, vì chúng có thể phải được di chuyển để tạo ra chỗ trống cho các phần tử nhỏ hơn được gặp sau đó. Tuy nhiên, mảng được sắp trọn vẹn khi con trỏ đi tới đầu bên phải.

Có một chi tiết quan trọng hơn phải xem xét: đó là thủ tục insertion không hoạt động, vì lệnh while sẽ chạy lố quá đầu bên trái của mảng khi v là phần tử nhỏ nhất trong mảng. Để sửa lỗi này, chúng ta đặt một khoá "cầm canh" trong a[0], làm cho nó ít nhất là nhỏ bằng phần tử nhỏ nhất trong mảng. Các khoá cầm canh được dùng phổ biến trong các trường hợp giống như trường hợp này để tránh phải thêm vào một phép kiểm tra (trong trường hợp này là j>1) mà nó hầu như luôn luôn xảy ra ở vòng lặp trong.

Nếu vì một lý do nào đó không thuận tiện để dùng một biến cầm canh và mảng thực sự phải có các biên [1..N], Pascal chuẩn không cung cấp một cách khác sạch sẽ, vì nó không có một lệnh and "có điều kiện": trường hợp này có vẻ cần đến phép kiểm tra while (j>1) and (a[j-1]>v), nhưng lệnh này không hoạt động vì ngay cả khi j=1, phần thứ hai của and sẽ được lượng giá và sẽ gây ra một phép truy xuất mảng vượt-khỏi-biên. Một lệnh goto khỏi vòng lặp có vẻ như được cần đến (một vài người lập trình thích nhảy tới các khoảng cách nào đó để tránh các lệnh goto, ví dụ như bằng cách thực hiện một hành động bên trong vòng lặp để bảo đảm là vòng lặp kết thúc. Trong trường hợp này, một lời giải như vậy có vẻ khó được biện minh, vì nó khiến cho chương trình không rõ ràng hơn và thêm vào các tổn phí mỗi lần đi qua vòng lặp để kiểm tra một biến cố hiếm).

BÀN THÊM: SẮP XẾP NỔI BỌT (Bubble Sort)

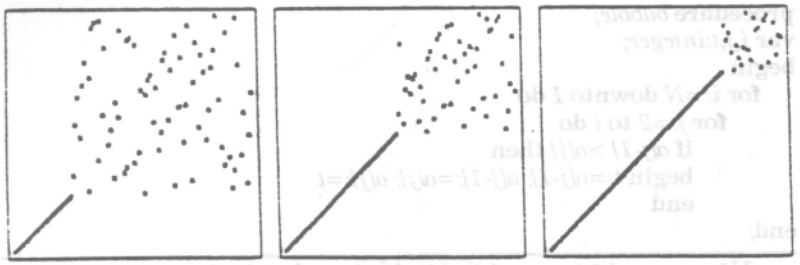
Một phương pháp sắp xếp cơ bản thường được dạy trong các lớp nhập môn là sắp xếp nổi bọt: duy trì việc di chuyển qua tập tin, hoán vị các phần tử kề nhau, nếu cần; khi không cần đến một hoán vị nào khác ở một bước nào đó thì tập tin được sắp xong. Một cài đặt của phương pháp này được cho dưới đây:

```
procedure bubble;
var i,j,t:integer;
begin
   for i:=N downto 1 do
      for j:=2 to i do
         if a[j-1] >a[j] then
         begin t:=a[j-1]; a[j-1]:=a[j]; a[j]:=t
         end
end;
```

Hãy suy nghĩ trong chốc lát để tin tưởng là đoạn chương trình này hoạt động được. Để làm điều này, chú ý là bất kỳ lúc nào phần tử lớn nhất được bắt gặp trong bước đầu tiên, nó sẽ được hoán vị với mỗi một phần tử nằm bên phải của nó, cho đến khi nó đi đến vị trí tận cùng bên phải của mảng. Sau đó, ở bước thứ hai, phần tử lớn nhất kế tiếp sẽ được đặt vào đúng vị trí,... Vì vậy sắp xếp nổi bọt thao tác như một kiểu sắp xếp chọn, mặc dù nó không làm nhiều việc hơn để đưa từng phần tử vào đúng vị trí.

ĐẶC TRƯNG VỀ HIỆU QUẢ CỦA CÁC PHÉP SẮP XẾP CƠ BẢN

Các minh họa trực tiếp về các đặc trưng thao tác của phép sắp chọn, sắp xếp chèn và sắp xếp nổi bọt được cho trong hình 8.3, 8.4 và 8.5. Các hình vẽ này cho thấy nội dung của mảng a đối với từng thuật toán sau khi vòng lặp ngoài đã lặp được N/4, N/2, và 3N/4 lần (bắt đầu bằng một phép xáo trộn ngẫu nhiên của các số nguyên từ 1 tới N coi như dữ kiện nhập cần sắp). Trong các hình vẽ, một hình vuông được đặt ở vị trí (i,j) với a[i]=j. Một mảng không có thứ tự vì vậy sẽ là sự hiển thị ngẫu nhiên của các hình vuông; trong một mảng đã được sắp, mỗi hình vuông sẽ xuất hiện ở trên cái nằm bên tay trái của nó. Để cho rõ ràng trong các hình vẽ, ta chỉ ra các phép chuyển vị (các phép sắp xếp lại của các số nguyên từ 1 tới N), mà, khi đã được sắp, sẽ có các hình vuông tất cả đều thẳng hàng dọc theo đường chéo chính. Các hình vẽ cho thấy các phương pháp khác nhau đã tiến triển hướng về mục tiêu này như thế nào.



Hình 8.3 *Sắp xếp chọn một chuyển vị ngẫu nhiên*

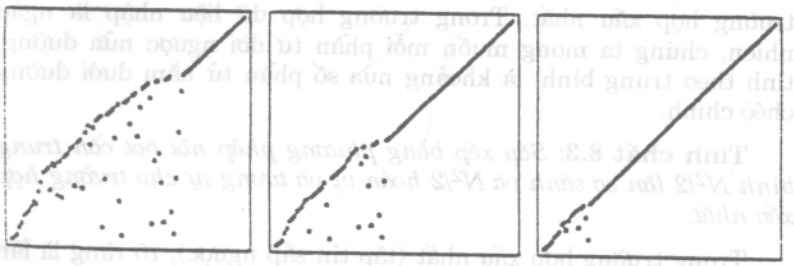
Hình 8.3 minh họa làm thế nào phép sắp xếp chọn chuyển từ trái sang phải, đặt các phần tử vào vị trí kết quả của chúng mà không nhìn ngược lại. Điều không rõ ràng trong đồ hình này sắp xếp chọn tốn nhiều thời gian để tìm phần tử nhỏ nhất trong phần chưa sắp của mảng.

Hình 8.4 trình bày cách thức phương pháp chèn đi từ trái qua phải, chèn các phần tử mới phát hiện vào đúng chỗ mà không cần quan sát gì thêm ở phía trước. Phía trái của mảng thay đổi liên tục.

Hình 8.5 cho thấy sắp xếp chọn và nổi bọt tương tự nhau. Sắp xếp bằng phương pháp nổi bọt "chọn" phần tử lớn nhất trong số còn lại trong mỗi giai đoạn, nhưng không tận dụng được thứ tự để sắp phần còn lại của mảng.

Tất cả các phương pháp đều tỉ lệ N^2 trong cả hai trường hợp tốt và xấu nhất và không đòi hỏi nhiều bộ nhớ. Vì vậy các lần so

Hình 8.4 *Sắp xếp chèn một chuyển vị ngẫu nhiên*



Hình 8.5 *Sắp xếp nổi bọt một chuyển vị ngẫu nhiên*

sánh của chúng phụ thuộc vào chiều dài của các vòng lặp trong và đặc tính của dữ liệu nhập.

Tính chất 8.1: *Sắp xếp bằng phương pháp chọn (selection sort) cần $N^2/2$ lần so sánh và N lần hoán vị.*

Tính chất này dễ thấy ở hình 8.1, là bảng NxN trong đó một ký tự tương ứng với một phép so sánh. Nhưng nó chỉ khoảng một nửa các phần tử nằm trên đường chéo. N-1 phần tử trên đường chéo (trừ phần tử cuối), mỗi phần tử tương ứng với một hoán vị. Chính xác hơn: Với mọi i từ 1 đến N-1, có một hoán vị và N-i so sánh, vì thế tổng cộng có N-1 hoán vị và (N-1)+(N-2)+...+2+1 = N(N-1)/2 so sánh. Dù cho dữ liệu nhập như thế nào đi nữa thì kết quả trên vẫn giữ nguyên: phần duy nhất của phương pháp sắp xếp chọn phụ thuộc vào việc nhập là số lần giá trị min được cập nhật. Trong trường hợp xấu nhất, các thuật toán cũng tỉ lệ N^2 nhưng trong trường hợp trung bình, tính toán này tỉ lệ với NlogN vì thế chúng ta mong muốn là thời gian chạy phương pháp sắp bằng chọn sẽ không bị ảnh hưởng bởi dữ liệu nhập.

Tính chất 8.2: *Sắp xếp bằng phương pháp chèn trung bình cần $N^2/4$ lần so sánh và $N^2/8$ lần hoán vị, xấu nhất gấp đôi số lần này.*

Như ở trên đã cài đặt, số lần so sánh và "hoán vị một nửa" (di chuyển) ngang nhau. Việc tính toán này dễ quan sát ở hình 8.2. Hình vẽ NxN cho thấy chi tiết của thao tác của thuật toán. Số phần tử bên dưới đường chéo chính đếm được là các phần tử trong

trường hợp xấu nhất. Trong trường hợp dữ liệu nhập là ngẫu nhiên, chúng ta mong muốn mỗi phần tử dời ngược nửa đường, tính theo trung bình, là khoảng nửa số phần tử nằm dưới đường chéo chính.

Tính chất 8.3: *Sắp xếp bằng phương pháp nổi bọt cần trung bình $N^2/2$ lần so sánh và $N^2/2$ hoán vị và tương tự cho trường hợp xấu nhất.*

Trong trường hợp xấu nhất (tập tin sắp ngược), rõ ràng là lần sắp xếp nổi bọt thứ i cần (N-i) so sánh và hoán vị, chứng minh giống như sắp bằng phương pháp chọn. Nhưng thời gian chạy của sắp bằng phương pháp nổi bọt phụ thuộc vào dữ liệu nhập. Chẳng hạn, chỉ cần một lần lặp khi tập tin đã sắp rồi (trong trường hợp này phương pháp chèn cũng nhanh như vậy). Trường hợp trung bình không tốt hơn bao nhiêu so với trường hợp xấu nhất mặc dù việc phân tích có vẻ phức tạp hơn.

Tính chất 8.4: *Sắp xếp bằng phương pháp chèn là tuyến tính đối với các tập tin "hầu như đã được sắp".*

Mặc dù quan điểm về tập tin "hầu như đã được sắp" là khá không chính xác, nhưng sắp bằng phương pháp chèn thao tác tốt trên một số kiểu tập tin không ngẫu nhiên thường có trong thực tế; thực ra sắp bằng phương pháp chèn đã lợi dụng được thứ tự hiện có trong tập tin.

Chẳng hạn, để ý thao tác trong sắp xếp bằng phương pháp chèn trên tập tin đã sắp. Mỗi phần tử sẽ được xác nhận ngay vị trí thích hợp của nó trên tập tin, và tổng thời gian thực hiện thì tuyến tính. Tương tự cho sắp bằng phương pháp nổi bọt, nhưng sắp bằng phương pháp chọn vẫn tỉ lệ N^2. Mặc dù tập tin không được sắp hoàn toàn, nhưng sắp bằng phương pháp chèn có thể rất có ích vì thời gian chạy phụ thuộc mạnh vào thứ tự hiện có trong tập tin. Thời gian chạy phụ thuộc vào số lần hoán chuyển: với mỗi phần tử, đếm số phần tử nằm bên trái lớn hơn nó. Đó là khoảng cách mà các phần tử cần dời đổi khi chèn vào tập tin trong quá trình sắp xếp. Tập tin có sẵn các phần có thứ tự thì ít phải chuyển đổi hơn.

Giả sử người ta muốn thêm vài phần tử vào một tập tin đã sắp để tạo một tập tin lớn hơn có thứ tự. Một cách để làm điều này là

thêm các phần tử mới vào cuối tập tin sau đó gọi một thuật toán sắp xếp. Rõ ràng là số lần hoán chuyển sẽ thấp trên tập tin này: một tập tin có một số không đổi các phần tử chưa đặt đúng chỗ sẽ chỉ dùng một số lần hoán chuyển tuyến tính. Một ví dụ khác là tập tin trong đó mỗi phần tử cách vị trí cuối cùng của nó (vị trí kết quả) một khoảng không đổi. Các tập tin như vậy có thể được tạo trong các bước khởi tạo của một vài thuật toán sắp xếp cải tiến: theo một quan điểm nào đó thì nó cũng chính là một dạng của sắp xếp bằng phương pháp chèn.

Để so sánh thêm giữa các phương pháp, người ta cần phân tích chi phí của các phép so sánh và hoán vị, một yếu tố phụ thuộc vào kích thước của các mẫu tin và khoá. Chẳng hạn, nếu mẫu tin là các khoá lưu một từ (word) thì một hoán vị (hai lần truy xuất mảng) sẽ tốn hai lần so với so sánh. Trong tình huống này, thời gian chạy của sắp bằng phương pháp chọn và phương pháp chèn có thể so sánh sơ lược, nhưng sắp bằng phương pháp nổi bọt thì chậm hơn hai lần (thường phương pháp này chậm hơn hai lần trong mọi trường hợp !). Nhưng nếu các mẫu tin cần nhiều so sánh trên khoá, phương pháp sắp xếp chọn sẽ tốt hơn.

Tính chất 8.5: *Sắp xếp bằng phương pháp chọn là tuyến tính đối với các tập tin có mẫu tin kích thước lớn và khoá kích thước nhỏ.*

Giả sử chi phí cho một lần so sánh là một đơn vị thời gian và chi phí cho một lần hoán vị là M đơn vị thời gian (ví dụ như trường hợp ứng với các mẫu tin M từ và khoá 1-từ). Lúc này, sắp xếp bằng phương pháp chèn cần N^2 lần so sánh và NM lần hoán vị để sắp một tập tin có kích thước NM. Nếu N tỉ lệ với M, số lượng dữ liệu là tuyến tính.

SẮP XẾP TẬP TIN CÓ MẪU TIN KÍCH THƯỚC LỚN

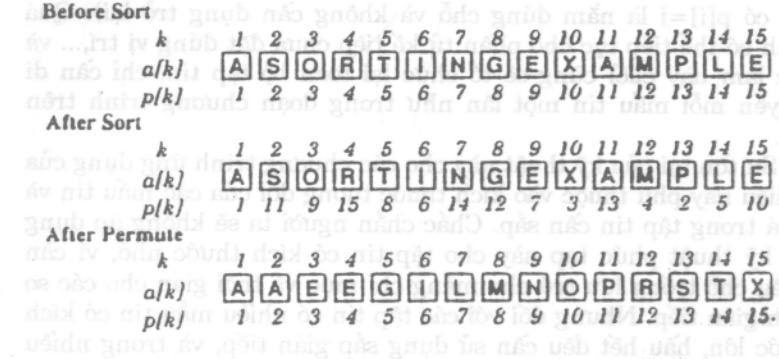
Bất kỳ phương pháp sắp xếp nào chỉ cần N lần hoán vị toàn bộ mẫu tin. Điều này có thể thực hiện được bằng cách cho thuật toán thao tác gián tiếp trên tập tin và sau đó tổ chức lại. Điển hình, nếu mảng a[1..N] chứa các mẫu tin có kích thước lớn, sau đó thao tác trên "mảng con trỏ" p[1..N], mảng này truy xuất mảng nguyên

thủy chỉ dùng để so sánh. Nếu đầu tiên định nghĩa p[i] = i, thuật toán trên (và cả những thuật toán sau này) chỉ cần sửa đổi thành a[p[i]] hơn là a[i] khi dùng a[i] để so sánh, và đề cập đến p hơn là a trên các thao tác dữ liệu. điều này đưa ra một thuật toán sẽ "sắp" mảng chỉ mục sao cho p[1] là chỉ mục của phần tử nhỏ nhất trên a, p[2] là chỉ mục của phần tử nhỏ thứ hai trên a,..., và cần tránh chi phí cho việc di chuyển các mẩu tin có kích thước lớn. Đoạn chương trình sau cho thấy sắp xếp bằng phương pháp chèn được bổ sung để làm việc theo cách này.

```
procedure insertion;
var i,j,v:integer;
begin
   for i:=1 to N do p[i]:=i;
   for i:=2 to N do
   begin v:=p[i]; j:=i;
      while a[p[j-1]] a[v] do
      begin p[j]:=p[j-1]; j:=j-1
      end;
      p[j]:=v;
   end
end;
```

Trong chương trình này, mảng a được truy xuất chỉ để so sánh các khoá của hai mẩu tin. Vì vậy, dễ sửa đổi để xử lý các tập tin có mẩu tin có kích thước lớn bằng cách sửa đổi phép so sánh để truy xuất chỉ một trường tin nhỏ của một mẩu tin lớn hay viết đoạn so sánh thành một thủ tục phức tạp hơn. Hình 8.6 cho thấy quá trình này sinh ra một sự chuyển vị xác định thứ tự trong đó các phần tử của mảng có thể được truy xuất để xác định một xâu được sắp. Trong nhiều ứng dụng, điều này là đủ (dữ liệu có thể không cần dời chuyển gì cả). Chẳng hạn, người ta có thể xuất dữ liệu ra theo một thứ tự đã được sắp chỉ bằng cách tham khảo gián tiếp qua mảng chỉ mục, cứ như là tự nó đã sắp rồi.

Nhưng điều gì xảy ra khi dữ liệu thực sự cần sắp lại như ở cuối hình 8.6 ? Nếu có đủ bộ nhớ cho một bản sao khác của mảng, thì điều này là tầm thường, nhưng trong trường hợp thông thường hơn thì vấn đề như thế nào khi không có đủ chỗ cho một bản sao khác của tập tin ?



Hình 8.6 *Tổ chức lại một mảng "đã được sắp"*

Trong ví dụ, A đầu tiên được đặt vào vị trí thích hợp với p[1]=1 vì thế không cần phải làm gì cả. Việc đầu tiên cần làm là đặt mẫu tin cùng với khoá nhỏ nhất kế tiếp (có chỉ mục p[2]) vào vị trí thứ hai trên tập tin. Nhưng trước khi làm điều này, cần cất mẫu trong vị trí đó, gọi là t. Rồi sau khi dời chuyển, chúng ta nhận ra là có một "lỗ hổng" (hole) trong tập tin ở vị trí p[2]. Nhưng chúng ta biết là mẫu tin ở vị trí p[p[2]] cần lấp "lỗ hổng" này. Bằng cách này, chúng ta đi đến quan điểm là cần lưu giữ phần tử nguyên thủy ở vị trí thứ hai. Điều này dẫn đến một số các phép gán trong ví dụ chúng ta đang xem xét: t:=a[2]; a[2]:=a[11]; a[11]:=a[13]; a[13]:=a[2]; a[2]:=t. Các phép gán này đặt các mẫu tin có khoá A,P và S vào vị trí thích hợp trong tập tin mà nó có thể được đánh dấu do p[2]=2, p[11]=11 và p[13]=13 (bất kỳ phần tử

```
procedure insitu;
var i,j,k,t: integer;
begin
  for i:=1 to N do if p[i]<>i then
  begin
    t:=a[i]; k:=i;
    repeat j:=k; a[j]:=a[p[j]]; k:=p[j]; p[j]:=j;
    until k=i;
    a[j]:=t
  end;
end;
```

nào có p[i]=i là nằm đúng chỗ và không cần đụng trở lại). Quá trình có thể tiếp tục cho phần tử kế tiếp chưa đặt đúng vị trí,... và việc làm này cuối cùng sẽ tổ chức lại toàn bộ tập tin, chỉ cần di chuyển mỗi mẩu tin một lần như trong đoạn chương trình trên đây.

Sự tồn tại của kỹ thuật này cho các chương trình ứng dụng của tài liệu này phụ thuộc vào kích thước tương đối của các mẩu tin và khoá trong tập tin cần sắp. Chắc chắn người ta sẽ không áp dụng các kỹ thuật phức tạp này cho tập tin có kích thước nhỏ, vì cần nhiều chỗ trống lưu trữ cho mảng chỉ mục và thời gian cho các so sánh gián tiếp. Nhưng đối với các tập tin có nhiều mẩu tin có kích thước lớn, hầu hết đều cần sử dụng sắp gián tiếp, và trong nhiều chương trình ứng dụng không cần thiết di chuyển dữ liệu gì cả. Dĩ nhiên, đối với các tập tin có mẩu tin kích thước lớn, cần sử dụng sắp xếp bằng phương pháp chọn như đã trình bày ở trên.

Vì tính khả dụng của cách tiếp cận gián tiếp này, trong chương này và các chương sau chúng ta đi đến kết luận là có khả năng ứng dụng các phương pháp sắp xếp cho các trường hợp tổng quát hơn sau khi so sánh các phương pháp sắp xếp các tập tin chứa số nguyên.

SẮP XẾP BẰNG PHƯƠNG PHÁP SHELLSORT

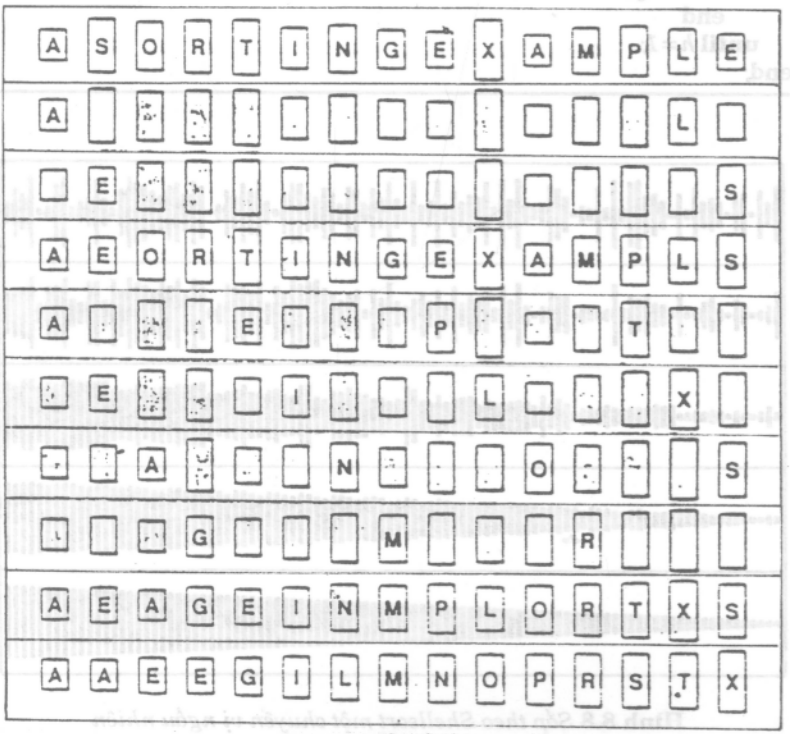
Sắp xếp bằng phương pháp chèn khá chậm vì nó chỉ hoán vị các phần tử kề nhau. Ví dụ, nếu phần tử nhỏ nhất nằm ở cuối mảng, cần thực hiện N lần để đến vị trí cuối cùng của nó. Shellsort là mở rộng đơn giản của sắp xếp bằng phương pháp chèn, phương pháp này đạt được tốc độ bằng cách hoán vị giữa các phần tử ở xa nhau.

Ý tưởng là sắp xếp lại tập tin để cho nó có tính chất là việc lấy mọi phần tử thứ h (bắt đầu bất kỳ vị trí nào) cũng đều cho ra một tập tin đã sắp. Một tập tin như vậy được gọi là sắp theo độ dài bước h. Nói một cách khác, một tập tin được sắp theo độ dài bước h là h tập tin được sắp độc lập, đan xen với nhau. Bằng cách sắp xếp theo độ dài bước h ứng với vài giá trị h khá lớn, chúng ta có thể di chuyển các phần tử ở những khoảng cách xa trong mảng và vì vậy

dễ dàng hơn để sắp xếp độ dài bước h cho các giá trị h nhỏ hơn. Dùng thủ tục này cho bất kỳ một dãy các giá trị của h tận cùng là 1 sẽ cho ra một tập tin đã sắp xong: đây là Shellsort.

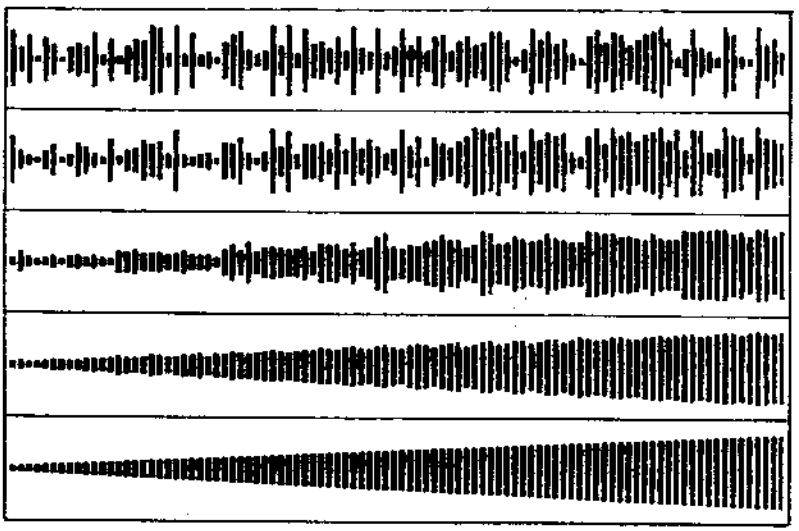
Hình 8.7 cho thấy thao tác của Shellsort trên tập tin ví dụ ứng với các bước 13, 4, 1. Trong lần lặp đầu tiên, A ở vị trí 1 được so sánh với L ở vị trí 14, sau đó S ở vị trí 2 được so sánh (và hoán vị) với E ở vị trí 15. Trong lần lặp thứ hai, A T E P ở vị trí 1,5,9 và 13 được sắp xếp lại để đặt A E P T vào các vị trí đó, và tương tự cho các vị trí 2,6,10 và 14,... Lần lặp cuối chỉ là sắp bằng chèn, nhưng không có phần tử nào phải dời chuyển rất xa.

Một cách để cài đặt Shellsort là, ứng với mỗi giá trị h, sử dụng chèn trực tiếp một cách độc lập trên từng tập tin trong số h tập tin con (Phần tử cầm canh sẽ không được dùng vì đó sẽ phải là h



Hình 8.7 *Shellsort*

```
program shellsort;
label O;
var i,j,h,v: integer;
begin
   h:=1;
   repeat h:=3*h+1 until h>N;
   repeat
      h:=h div 3;
      for i:=h+1 to N do
      begin v:=a[i]; j:=i;
         while a[j-h] > v do
         begin a[j]:=a[j-h]; j:=j-h;
               if j<=h then goto O
         end;
         O: a[j]:=v
      end
   until h=1;
end;
```



Hình 8.8 *Sắp theo Shellsort một chuyển vị ngẫu nhiên*

ứng với giá trị lớn nhất h được dùng). Nhưng dễ hơn nhiều là: nếu chúng ta thay thế mọi sự xuất hiện của "1" bằng "h" ("2" bằng "h+1") trong sắp xếp bằng chèn, thì chương trình sắp xếp tập tin theo độ dài bước h sẽ như trang trước:

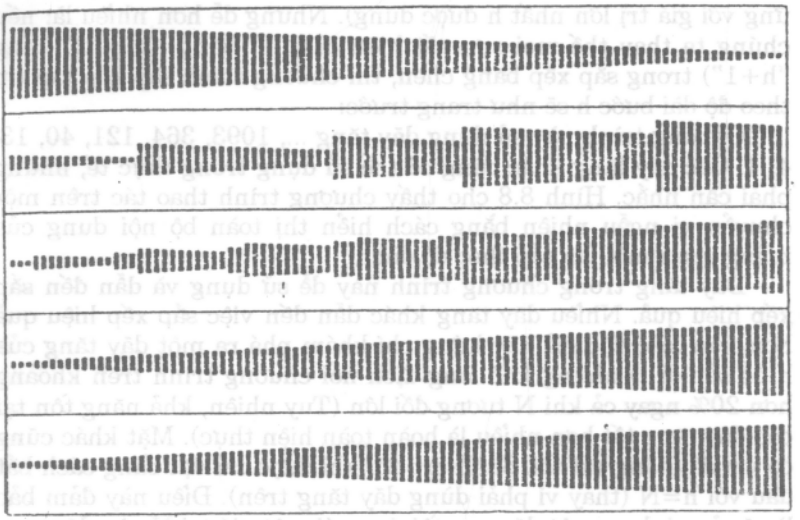
Chương trình này sử dụng dãy tăng ..., 1093, 364, 121, 40, 13, 4, 1. Các dãy tăng khác cũng có thể sử dụng trong thực tế, nhưng phải cân nhắc. Hình 8.8 cho thấy chương trình thao tác trên một chuyển vị ngẫu nhiên bằng cách hiển thị toàn bộ nội dung của mảng a sau mỗi lần sắp theo độ dài h.

Dãy tăng trong chương trình này dễ sử dụng và dẫn đến sắp xếp hiệu quả. Nhiều dãy tăng khác dẫn đến việc sắp xếp hiệu quả hơn nữa (các độc giả hãy hứng chí khám phá ra một dãy tăng của h để sắp đi !) nhưng khó lòng địch nổi chương trình trên khoảng hơn 20% ngay cả khi N tương đối lớn (Tuy nhiên, khả năng tồn tại các dãy tăng tốt hơn nhiều là hoàn toàn hiện thực). Mặt khác cũng có vài dãy tăng rất tồi. Shellsort đôi khi được cài đặt bằng cách bắt đầu với h=N (thay vì phải dùng dãy tăng trên). Điều này đảm bảo là sẽ sản sinh ra một dãy con tồi ứng với một giá trị N nào đó.

Mô tả trên về độ hiệu quả của Shellsort là không chính xác bởi vì không có ai có thể phân tích thuật toán. Điều này dẫn đến khó khăn không những khi đánh giá các dãy tăng khác nhau mà còn khi so sánh Shellsort với các phương pháp khác về mặt giải tích. Thậm chí dạng hàm của thời gian chạy cả thời gian chạy cho Shellsort cũng không biết được (Hơn thế nữa, dạng hàm phụ thuộc vào dãy tăng). Đối với chương trình trên, phỏng chừng là $N(\log N)^2$ và $N^{1.25}$. Thời gian chạy không bị ảnh hưởng bởi thứ tự ban đầu của tập tin, nhưng ngược lại sắp xếp bằng chèn trực tiếp lại tuyến tính với tập tin sắp sẵn nhưng tỉ lệ N^2 với tập tin sắp ngược. Hình 8.9 trang sau minh họa thao tác của Shellsort trên một tập tin theo thứ tự ngược.

Tính chất 8.6: *Shellsort không bao giờ thực hiện hơn $N^{3/2}$ so sánh (đối với dãy tăng 1,4,13,40,121, ...)*

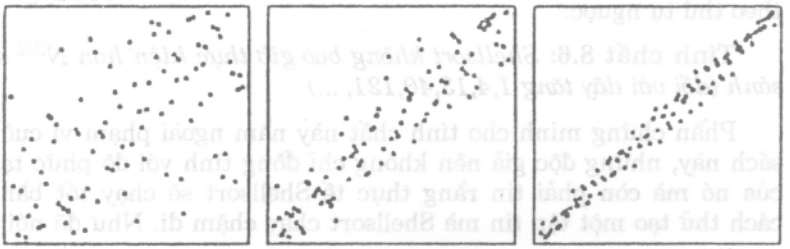
Phần chứng minh cho tính chất này nằm ngoài phạm vi cuốn sách này, nhưng độc giả nên không chỉ đồng tình với độ phức tạp của nó mà còn phải tin rằng thực tế Shellsort sẽ chạy tốt bằng cách thử tạo một tập tin mà Shellsort chạy chậm đi. Như đã nói ở



Hình 8.9 *Sắp theo Shellsort một chuyển vị thứ-tự-ngược*

trên, có vài dãy tăng không hiệu quả mà Shellsort cần một số bình phương lần so sánh, nhưng giới hạn $N^{3/2}$ đã được chứng minh là đúng đối với một phạm vi rộng các dãy, gồm cả dãy đã dùng ở trên.

Hình 8.10 cho thấy hình ảnh của thao tác trong Shellsort, có thể được so sánh với các hình 8.3, 8.4, 8.5. Nó cho thấy nội dung của mảng sau mỗi lần sắp (trừ lần cuối cùng hoàn thành việc sắp xếp). Trong hình vẽ này chúng ta có thể hình dung ra một dải cao su bị đồn cục lại ở các điểm trải dài ở góc dưới trái và trên phải, bị



Hình 8.10 *Sắp theo Shellsort một chuyển vị ngẫu nhiên*

kéo căng hơn để đẩy tất cả các điểm hướng về phía đường chéo. Ba hình 8.3, 8.4 và 8.5 mỗi đồ hình biểu diễn một khối lượng công việc đáng kể của từng thuật toán đã trình bày, ngược lại mỗi đồ hình trong hình 8.10 biểu diễn chỉ một lần sắp xếp theo độ dài h.

Shellsort là phương pháp được chọn cho nhiều ứng dụng sắp xếp bởi vì nó có thời gian chạy có thể chấp nhận được ngay cả đối với tập tin có kích thước lớn vừa phải (ít hơn 5000 phần tử) và nó đòi hỏi chỉ một lượng mã chương trình rất nhỏ dễ hoạt động. Chúng ta xem xét các phương pháp hiệu quả hơn nữa trong vài chương kế nhưng có lẽ chỉ nhanh hơn hai lần ngoại trừ N lớn, và chúng phức tạp hơn đáng kể. Tóm lại, nếu bạn có một bài toán sắp xếp, thì hãy dùng chương trình trên, sau đó xác định xem khi nào việc bỏ công sức ra để thay nó bằng một phương pháp tinh vi hơn là đáng để làm.

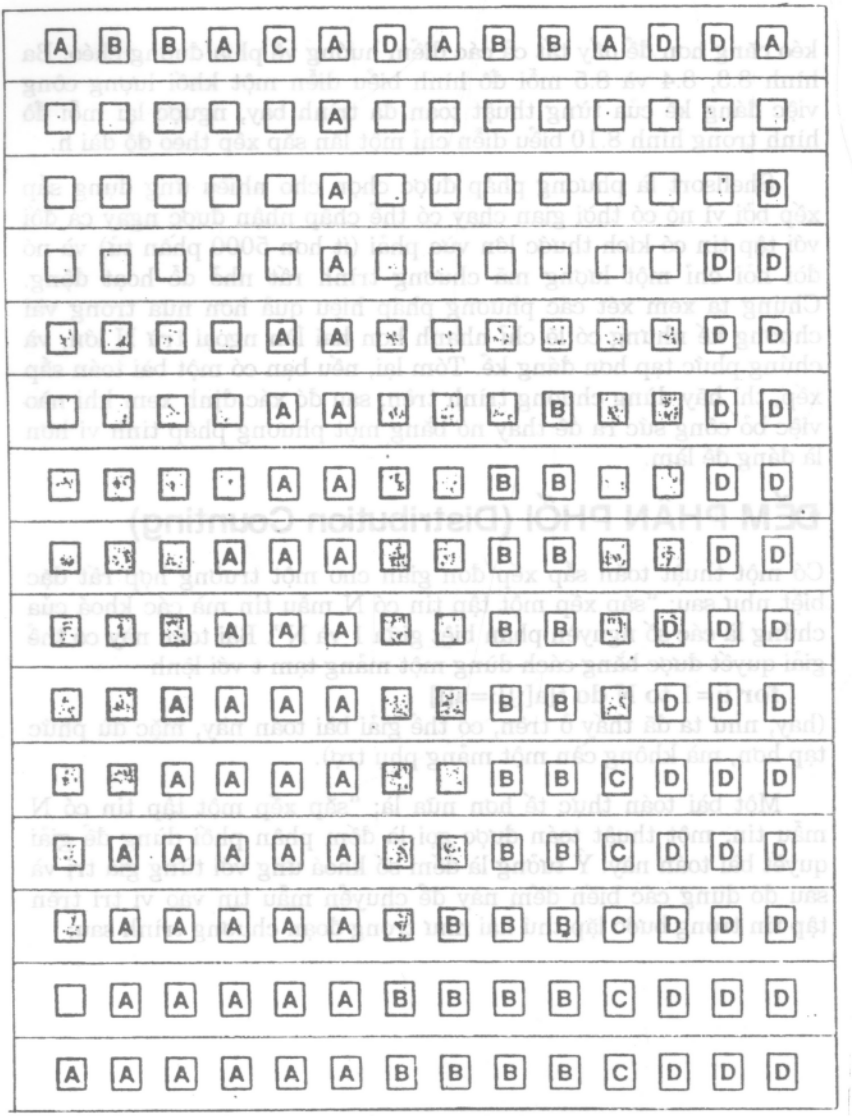
ĐẾM PHÂN PHỐI (Distribution Counting)

Có một thuật toán sắp xếp đơn giản cho một trường hợp rất đặc biệt như sau: "sắp xếp một tập tin có N mẫu tin mà các khoá của chúng là các số nguyên phân biệt giữa 1 và N". Bài toán này có thể giải quyết được bằng cách dùng một mảng tạm t với lệnh

for i:=1 **to** N **do** t[a[i]]:=a[i]

(hay, như ta đã thấy ở trên, có thể giải bài toán này, mặc dù phức tạp hơn, mà không cần một mảng phụ trợ).

Một bài toán thực tế hơn nữa là: "sắp xếp một tập tin có N mẫu tin, một thuật toán được gọi là đếm phân phối dùng để giải quyết bài toán này. Ý tưởng là đếm số khoá ứng với từng giá trị và sau đó dùng các biến đếm này để chuyển mẫu tin vào vị trí trên tập tin trong bước lặp thứ hai như trong đoạn chương trình sau:



Hình 8.11 *Đếm phân phối*

```
for j:=0 to M-1 do count[j]:=0;
for i:=1 to N do count[a[i]]:=count[a[i]]+1;
for j:=1 to M-1 do count[j]:=count[j-1]+count[j];
for i:=N downto 1 do
   begin   b[count[a[i]]]:=a[i];
           count[a[i]]:=count[a[i]]-1
   end;
for i:=1 to N do a[i]:=b[i];
```

Để thấy đoạn chương trình này hoạt động như thế nào, xét tập tin ví dụ gồm các số nguyên ở hàng đầu tiên trong hình 8.11. Vòng lặp **for** đầu tiên khởi động các biến đếm bằng 0, vòng lặp thứ hai khởi động các biến đếm count[1]=6, count[2]=4, count[3]=1 và count[4]=4 bởi vì có 6 phần tử A, 4 phần tử B, ... Kế tiếp vòng lặp **for** thứ ba cộng các con số này lại với nhau để cho count[1]=6, count[2]=10, count[3]=11 và count[4]=15. Có 6 khoá nhỏ hơn hay bằng A, 10 khoá nhỏ hơn hay bằng B, ...

Bây giờ, các khoá này có thể dùng như địa chỉ để sắp xếp mảng như đã minh họa trong hình vẽ. Mảng nhập a nguyên thủy được chỉ ra ở dòng đầu; phần còn lại của hình cho thấy mảng tạm t đang được đổ đầy. Ví dụ như khi A ở cuối tập tin được phát hiện, nó được đặt vào vị trí 6 vì là count[1] nói lên là có 6 khoá nhỏ hơn hay bằng A. Sau đó D từ vị trí kế đến vị trí cuối cùng trong tập tin được đặt vào vị trí 14 và count[4] bị giảm đi,... Vòng lặp trong đi từ N xuống 1 để cho việc sắp xếp là ổn định.

Phương pháp này sẽ làm việc rất tốt đối với kiểu tập tin đã được tuyển chọn. Hơn nữa, nó có thể mở rộng được để cho ra một phương pháp mạnh hơn nhiều mà chúng ta sẽ xem xét trong chương 10.

BÀI TẬP

1. Hãy cho một dãy các thao tác "so sánh-hoán vị" để sắp xếp bốn mẩu tin.

2. Trong ba phương pháp sắp xếp cơ bản (phương pháp chọn, phương pháp chèn và phương pháp nổi bọt) thì phương pháp nào chạy nhanh nhất đối với một tập tin đã được sắp rồi ?

3. Trong ba phương pháp sắp xếp cơ bản thì phương pháp nào là nhanh nhất đối với một tập tin có thứ tự đảo ngược ?

4. Hãy kiểm chứng giả thiết phương pháp chọn là phương pháp nhanh nhất trong ba phương pháp cơ bản (để sắp xếp các số nguyên), sau đó tới sắp xếp chèn, và cuối cùng là sắp xếp nổi bọt.

5. Hãy đưa ra một lý do hợp lý tại sao không thuận tiện khi dùng một khoá cầm canh cho phương pháp sắp xếp chèn (trừ phương pháp phát sinh từ cài đặt của Shellsort) ?

6. Có bao nhiêu phép so sánh được dùng bởi Shellsort với sắp-7, rồi sắp-3 các khoá E A S Y Q U E S T I O N ?

7. Hãy cho một ví dụ để minh họa tại sao 8, 4, 2, 1 sẽ không là một cách tốt để kết thúc một dãy tăng của Shellsort ?

8. Phương pháp sắp xếp chọn có ổn định không ? Còn đối với sắp xếp chèn và sắp xếp nổi bọt thì thế nào ?

9. Hãy cho một phiên bản được chuyên biệt hoá của đếm phân phối để sắp các tập tin mà các phần tử chỉ có một trong hai giá trị (hoặc x hoặc y) ?

10. Hãy thử nghiệm với các dãy tăng khác nhau cho Shellsort: tìm một dãy mà nó chạy nhanh hơn dãy được cho bởi một tập tin ngẫu nhiên gồm 1000 phần tử.

9
QUICK SORT

Trong chương này, chúng ta sẽ nghiên cứu thuật toán sắp xếp có lẽ dùng rộng rãi hơn bất kỳ thuật toán nào khác, Quicksort là thuật toán cơ bản được A.R. Hoare phát minh vào 1960 và nhiều người đã nghiên cứu nó từ thời gian này. Quicksort là phương pháp phổ biến vì không khó cài đặt. Nó là phương pháp sắp xếp "hướng tổng quát" tốt (nó làm việc tốt trong các tình huống khác nhau) và trong nhiều trường hợp nó sử dụng ít tài nguyên hơn bất kỳ phương pháp sắp xếp khác.

Các ưu điểm của thuật toán Quicksort là chỉ sử dụng một ngăn xếp phụ nhỏ, chỉ cần khoảng trung bình NlogN thao tác để sắp N phần tử, và có một vòng lặp trong rất ngắn. Yếu điểm của nó là tính đệ qui, nó cần khoảng N^2 thao tác trong trường hợp xấu nhất và một lỗi đơn giản trong cài đặt có thể bị bỏ qua và có thể gây ảnh hưởng xấu cho vài tập tin.

Thuật toán Quicksort được đánh giá tốt, nhờ vào sự phân tích toán học và các khẳng định rất chính xác về tính năng của nó. Sự phân tích toán học đã được kiểm chứng qua thực nghiệm, và thuật toán đã được cải tiến để trở thành phương pháp chọn lọc trong các ứng dụng sắp xếp thực tế khác nhau. Điều này rất đáng để cho chúng ta quan sát thêm các thuật toán khác kỹ lưỡng hơn dựa vào cách cài đặt của Quicksort. Các kỹ thuật cài đặt tương tự rất thích hợp với các thuật toán khác, với Quick sort chúng ta có thể yên tâm sử dụng các kỹ thuật này vì quá trình thực hiện của nó được đánh giá rất tốt.

Người ta đã thử dựa vào các cách cải tiến Quicksort: một thuật toán sắp nhanh hơn là "bẫy chuột" (mousetrap). Từ lúc Hoare lần đầu tiên đưa ra thuật toán, các phiên bản "cải tiến" đần xuất hiện. Người ta đã thử và phân tích nhiều tư tưởng, nhưng dễ bị lừa vì

thuật toán cân bằng đến nỗi các hiệu quả cải tiến trong một phần của chương trình có thể bù đắp lại bằng hậu quả của việc thực hiện không tốt trong phần khác của chương trình. Chúng ta sẽ xem xét chi tiết về ba điều hiệu chỉnh để cải tiến Quicksort.

Một phiên bản được hiệu chỉnh kỹ lưỡng của Quicksort có lẽ chạy nhanh hơn bất kỳ phương pháp sắp xếp nào trên hầu hết các máy tính. Tuy nhiên nên đề phòng là việc hiệu chỉnh bất kỳ thuật toán nào có thể gây ra những kết quả ngoài ý muốn, bởi vì khi một ấn bản được phát triển thì có thể đây là chương trình dùng cho thư viện các chương trình tiện ích để sắp hay cho các ứng dụng sắp xếp quan trọng. Nhưng mặt khác nếu không có ai dám bỏ công ra để đảm bảo chắc là cài đặt của Quick không có khuyết điểm, thì Shellsort có thể là cách chọn lựa an toàn hơn ít phải tốn công cài đặt.

THUẬT TOÁN CƠ SỞ

Quicksort thuộc loại phương pháp "chia để trị" (divide and conquer). Nó thực hiện bằng cách phân hoạch một tập tin thành hai phần, sau đó sắp xếp các phần riêng biệt nhau. Như chúng ta đã biết, vị trí chính xác của các phần được phân hoạch phụ thuộc vào tập tin, vì thế thuật toán có cấu trúc đệ qui như sau:

```
procedure quicksort(l,r:integer);
var i:integer;
begin
   if rl then
   begin
      i:=partition(l,r);
      quicksort(l,i-1);
      quicksort(i+1,r);
    end
end;
```

Tham số l và r không giới hạn các tập tin con trong tập tin gốc cần sắp; lệnh gọi quicksort (1,N) sắp toàn bộ tập tin.

Yếu điểm chính của phương pháp này là thủ tục partition, tổ chức lại mảng thỏa ba điều kiện sau:

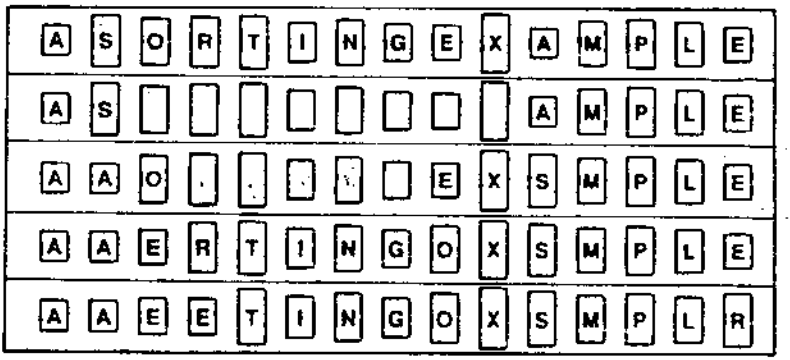
(i) phần tử a[i] đặt ở vị trí cuối cùng của nó trong mảng với i nào đó.

(ii) tất cả các phần tử trong a[l], ... , a[i-1] nhỏ hơn hay bằng a[i]

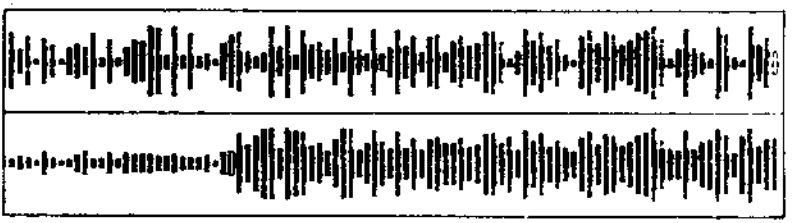
(iii) tất cả các phần tử trong a[i+1], ... a[r] lớn hơn hay bằng a[i]

Điều này có thể được cài đặt đơn giản và dễ dàng qua thuật toán tổng quát sau. Đầu tiên chọn tùy ý a[r] là phần tử sẽ rơi vào vị trí đặt cuối cùng của nó. Kế tiếp, quét từ trái của mảng cho đến khi gặp một phần tử lớn hơn a[r], và quét từ đầu phải cho đến khi gặp một phần tử nhỏ hơn a[r], Hai phần tử dừng việc quét dĩ nhiên không đúng vị trí trong mảng được phân hoạch cuối cùng nên phải hoán vị chúng (thực ra tốt nhất nên ngừng việc quét đối với những phần tử bằng a[r], dù là có thể đi vào một số hoán vị không cần thiết). Tiếp theo bảo đảm là tất cả các phần tử trên mảng ở bên trái con trỏ trái nhỏ hơn a[r] và các phần tử ở bên phải của con trỏ phải lớn hơn a[r]. Khi các con trỏ giao nhau, quá trình phân hoạch gần như hoàn tất, còn lại là hoán vị a[r] với phần tử trái nhất của tập tin con bên phải (phần tử do con trỏ trái trỏ đến).

Hình 9.1 cho thấy ví dụ về tập tin có các khóa được phân hoạch bằng phương pháp này. Phần tử phải nhất là E được chọn như là phần tử để phân hoạch. Đầu tiên việc quét từ phía trái sẽ dừng ở S, kết đó việc quét từ phía phải dừng ở A (như đã trình bày ở dòng thứ hai của bảng), kế tiếp hai phần tử được hoán vị cho nhau. Tiếp đến việc quét từ phía trái sẽ dừng ở O, từ phía phải dừng ở E (dòng thứ ba của bảng) và lại hoán vị hai phần tử này.



Hình 9.1 *Phân hoạch*



Hình 9.2 *Phân hoạch một tập tin lớn*

Rồi kế tiếp là các con trỏ giao nhau. Việc quét từ phía trái sẽ ngừng ở R và việc quét từ phía phải sẽ ngừng ở E. Hoán vị E ở phía phải với R, tập tin được phân chia còn lại ở dòng cuối của hình 9.1.

Dĩ nhiên quá trình phân chia không ổn định vì khóa được dịch chuyển qua một số lớn các khóa giống nó trong bất kỳ hoán vị nào.

Hình 9.2 cho thấy kết quả của một tập tin lớn hơn: có các phần tử nhỏ ở bên trái và lớn ở bên phải, tập tin được chia có "thứ tự" đáng kể hơn tập tin ngẫu nhiên. Kết thúc sắp xếp bằng cách sắp hai tập tin con ở mỗi phía của thuật toán phân hoạch nhờ vào dùng đệ qui. Chương trình sau đưa ra một bản cài đặt đầy đủ của phương pháp nầy.

```
procedure quicksort(l,r:integer);
var v,t,i,j:integer;
begin
   if r>l then
      begin v:=a[r]; i:=l-1; j:=r;
         repeat
            repeat i:=i+1 until a[i]=v;
            repeat j:=i-1 until a[j]<=v;
            t:=a[i]; a[i]:=a[j]; a[j]:=t;
         until j<i;
         a[j]:=a[i]; a[i]:=a[r]; a[r]:=t;
         quicksort(l,i-1); quicksort(i+1,r);
      end
end;
```

Trong cài đặt này, biến v lưu giữ vị trí hiện hành của "phần tử phân hoạch" a[r] và i, j là các con trỏ quét phía trái và phải của mảng. Hoán vị giữa a[i] và a[j] xảy ra với j<i ngay sau khi các con

trỏ giao nhau nhưng trước khi phát hiện sự giao nhau và thoát khỏi vòng lặp repeat bên ngoài. Ba lệnh gán theo sau vòng lặp này để cài đặt hoán vị a[i] với a[j] và a[i] với a[r] (để đặt phần tử phân hoạch vào đúng vị trí)

Giống như sắp xếp chèn, cần một khóa cầm canh để ngưng việc quét trong trường hợp phần tử phân hoạch là phần tử lớn nhất trong tập tin, vì chính phần tử phân hoạch ở phía phải của tập tin dùng để ngừng việc quét. Chúng ta sẽ khảo sát nhanh một cách thức để loại trừ hai phần tử linh canh.

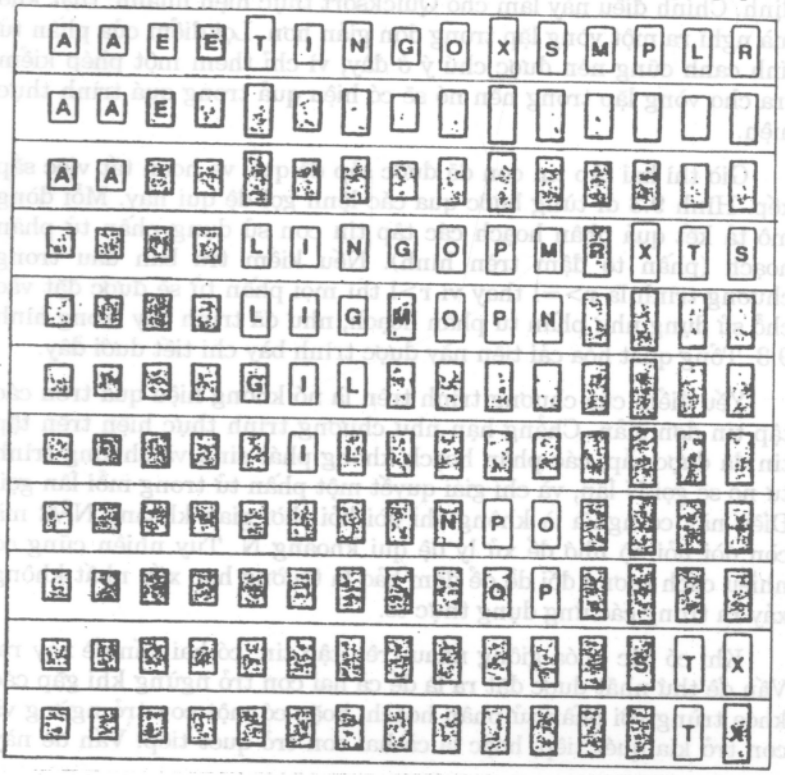
"Vòng lặp trong" của Quicksort liên quan đến việc tăng đơn giản một con trỏ và so sánh một phần tử mảng dựa vào giá trị cố định. Chính điều này làm cho Quicksort thực hiện nhanh: thật khó mà nghĩ ra một vòng lặp trong đơn giản hơn. Lợi điểm của phần tử lính canh cũng nên được chú ý ở đây, vì chỉ thêm một phép kiểm tra cho vòng lặp trong nên nó sẽ có hiệu quả trong quá trình thực hiện.

Giờ thì hai tập tin con đã được sắp đệ qui, và hoàn tất việc sắp xếp. Hình 9.3 đi từng bước qua các lệnh gọi đệ qui này. Mỗi dòng mô tả kết quả phân hoạch các tập tin con sử dụng phần tử phân hoạch (phần tô đậm trên hình). Nếu kiểm tra ban đầu trong chương trình là r>=l thay vì r>l thì mọi phần tử sẽ được đặt vào chỗ sử dụng như phần tử phân hoạch, như đã trình bày trong hình 9.3. Tổng quát hóa cải tiến này được trình bày chi tiết dưới đây.

Yếu điểm của chương trình trên là nó không hiệu quả trên các tập tin đơn giản. Chẳng hạn như chương trình thực hiện trên tập tin đã được sắp, các phân hoạch không phát sinh và chương trình tự nó sẽ gọi N lần, và chỉ giải quyết một phần tử trong mỗi lần gọi. Điều này có nghĩa là không chỉ đòi hỏi thời gian khoảng $N^2/2$ mà còn đòi hỏi bộ nhớ để xử lý đệ qui khoảng N. Tuy nhiên cũng có nhiều cách tương đối dễ để đảm bảo là trường hợp xấu nhất không xảy ra trong các ứng dụng thực tế.

Khi có các khóa giống nhau trên tập tin, có hai vấn đề xảy ra. Vấn đề thứ nhất được đặt ra là để cả hai con trỏ ngừng khi gặp các khóa trùng với phần tử phân hoạch, hoặc có một con trỏ ngừng và con trỏ kia quét tiếp, hoặc là cả hai con trỏ quét tiếp. Vấn đề này

thực sự đã được nghiên cứu chi tiết bằng toán học, và kết quả cho thấy tốt nhất là nên cho cả hai con trỏ dừng lại. Điều này hướng tới việc cân bằng các phân hoạch khi xuất hiện nhiều khóa trùng nhau. Thứ hai, vấn đề đặt ra là xử lý thích hợp con trỏ giao nhau khi có nhiều khóa trùng: thực ra chương trình ở trên có thể cải tiến sơ bộ bằng cách kết thúc việc quét khi j<i và sau đó dùng quicksort(l, j) cho lần gọi đệ qui đầu tiên. Đây là một cải tiến vì khi j=i chúng ta có thể đặt hai phần tử vào vị trí theo phân hoạch bằng cách để vòng lặp lặp thêm một lần nữa (trường hợp này xảy ra như đối với R và E trong ví dụ trên). Có lẽ nên tạo sự thay đổi như thế này vì chương trình đã nêu để lại một mẩu tin có khóa bằng với khóa phân hoạch a[r], và điều nầy làm cho phép phân



Hình 9.3 *Các tập tin con trong quicksort*

hoạch đầu tiên trong lệnh gọi quicksort (i+1,r) bị thoái hóa bởi vì khóa phải nhất của phân hoạch là khóa nhỏ nhất.

Cài đặt của phép phân hoạch đã nêu dễ hiểu hơn một ít; tuy nhiên có thể thay đổi cài đặt này khi có một số lượng lớn khóa trùng nhau.

ĐẶC TRƯNG VỀ HIỆU QUẢ CỦA QUICKSORT

Điều tốt nhất có thể xảy ra trong Quicksort là mỗi giai đoạn phân hoạch phân chia tập tin thành hai nửa. Điều nầy khiến cho số lần so sánh cần thiết cho Quicksort thỏa mãn công thức truy hồi "chia để trị" sau đây

$$C_N = 2C_{N/2} + N.$$

(đại lượng $2C_{N/2}$ là phí tổn của việc sắp xếp hai tập tin con; N là phí tổn của việc kiểm tra mỗi phần tử, dùng một con trỏ phân hoạch hay con trỏ khác). Từ chương 6, chúng ta đã biết công thức nầy có lời giải là

$$C_N \# N\lg N$$

Thực ra về mặt trung bình việc phân hoạch rơi vào phần tử giữa. Sự đánh giá xác suất chính xác của mỗi vị trí phân hoạch làm cho công thức truy hồi khó giải hơn nhưng kết quả cuối cùng vẫn giống nhau.

TÍNH CHẤT 9.1: *Về mặt trung bình, Quicksort chỉ cần 2NlnN so sánh.*

Công thức tính truy hồi chính xác để tính số lần so sánh mà Quicksort cần để hoán vị ngẫu nhiên N phần tử là:

$$C_N = N + 1 + \frac{1}{N}\sum_{1\leqslant k\leqslant N}(C_{k-1} + C_{N-k})$$

với $N>=2$ và $C_0=C_1=1$.

Giá trị N+1 bao hàm chi phí so sánh phần tử phân hoạch với mỗi phần tử còn lại, tổng còn lại mang ý nghĩa là mỗi phần tử k có thể là phần tử phân hoạch với xác suất 1/k và sau đó còn lại các tập tin kích thước k-1 và N-k

Mặc dù dường như quá phức tạp, nhưng công thức truy hồi nầy có thể giải dễ dàng theo ba bước. Đầu tiên,

$C_0 + C_1 + \ldots + C_{N-1}$ thì y hệt như $C_{N-1} + C_{N-2} + \ldots + C_0$

vì vậy ta có :

$$C_N = N + 1 + \frac{2}{N} \sum_{1 \leqslant k \leqslant N} C_{k-1}$$

Thứ hai, chúng ta có thể giản lược tổng này bằng cách nhân cả hai vế của nó với N và trừ đi $(N-1)C_{N-1}$:

$$NC_N - (N-1)C_{N-1} = N(N+1) - (N-1)N + 2C_{N-1}$$

Giản lược tiếp ta được: $NC_N = (N+1)C_{N-1} + 2N$

Thứ ba, chia cả hai vế cho $N(N+1)$ để có :

$$\frac{C_N}{N+1} = \frac{C_{N-1}}{N} + \frac{2}{N+1} = \frac{C_{N-2}}{N-1} + \frac{2}{N} + \frac{2}{N+1} = \ldots = \frac{C}{3} + \sum_{3 \leqslant k \leqslant N} \frac{2}{k+1}$$

Đáp số chính xác gần bằng với tổng số xấp xỉ bởi một tích phân :

$$\frac{C_N}{N+1} \# 2 \sum_{1 \leqslant k \leqslant N} \frac{1}{k} \# 2 \int_1^N \frac{1}{x} dx = 2\ln N$$

Chú ý là $2N\ln N \# 1.38N\lg N$ cho nên số lần so sánh trung bình chỉ cao hơn khoảng 38% trong trường hợp tốt nhất.

Vì vậy, cài đặt ở trên thực hiện rất tốt đối với các tập tin ngẫu nhiên nên là phương pháp sắp xếp tổng quát hợp lý trong nhiều chương trình ứng dụng. Tuy nhiên, nếu sử dụng phương pháp sắp xếp nhiều lần, hay dùng nó để sắp các tập tin kích thước lớn, thì vài cải tiến được nêu dưới đây sẽ làm cho trường hợp xấu sẽ ít xảy ra, giảm thời gian chạy trung bình xuống khoảng 20% và không cần dùng khóa lính canh.

KHỬ ĐỆ QUI

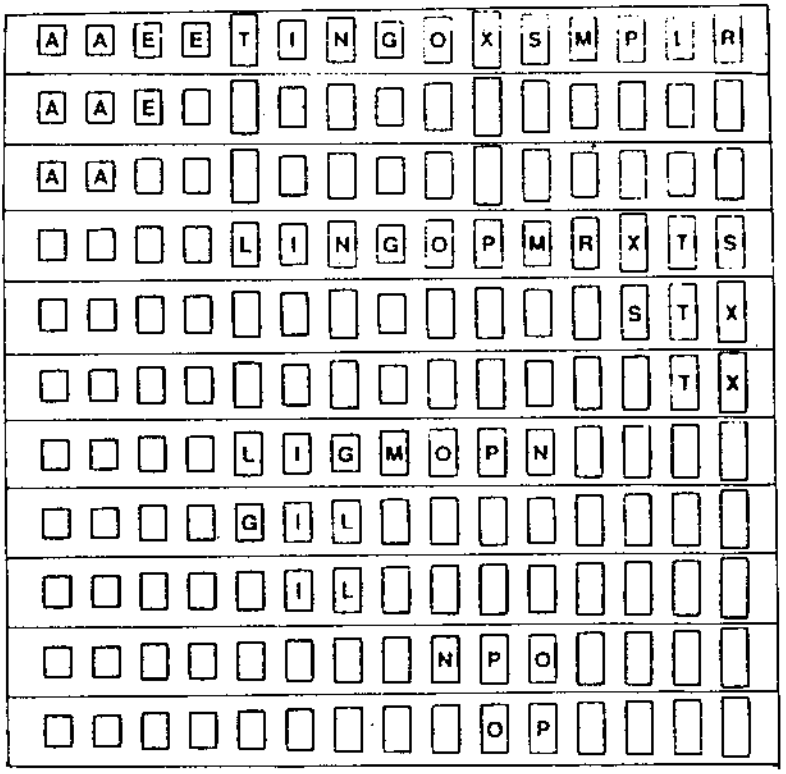
Giống như trong chương 5, Chúng ta có thể khử đệ qui trong chương trình Quicksort bằng cách dùng một ngăn xếp, lúc nào cần một tập tin con để xử lý, chúng ta lại lấy ra khỏi ngăn xếp. Khi phân hoạch, chúng ta tạo hai tập tin con có thể được đẩy vào trong

ngăn xếp. Điều này dẫn đến một cài đặt không đệ qui của Quicksort.

```
procedure quicksort;
   var t,i,l,r:integer;
   begin l:=1; r:=N; stackinit;
   push(l); push(r);
   repeat
      if r>l then
         begin
         i:=partition(l,r);
         if (i-l)>(r-i)
            then begin push(l); push(i-1); l:=i+1; end
            else begin push(i+1); push(r); r:=i-1; end;
         end
      else
         begin r:=pop; l:=pop end;
   until stackempty;
   end;
```

Chương trình này khác mô tả trên ở hai chỗ quan trọng. Thứ nhất, hai tập tin con không đặt trên ngăn xếp theo thứ tự tùy ý, nhưng kích thước của chúng sẽ được kiểm tra và tập tin nào có kích thước lớn hơn sẽ được đặt vào trong ngăn xếp trước. Thứ hai là tập tin nào có kích thước nhỏ hơn trong hai tập tin con không được đặt vào ngăn xếp; giá trị của các tham số được khởi động lại. Đây chính là kỹ thuật "khử đệ qui phần cuối" đã được thảo luận trong chương 5. Đối với Quicksort sự tổ hợp giữa kỹ thuật khử đệ qui này và cơ chế xử lý tập tin có kích thước nhỏ hơn bảo đảm là ngăn xếp chỉ cần chỗ trống cho khoảng lgN đầu vào, bởi vì mỗi đầu vào trên ngăn xếp sau đầu vào trên cùng phải đại diện cho một tập tin con có kích thước nhỏ hơn một nửa so với đầu vào phía trước.

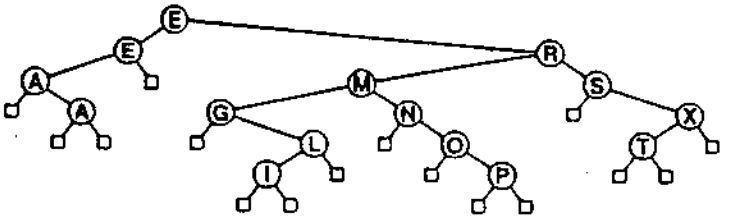
Trong trường hợp xấu nhất, điều này lại ngược lại với kích thước của ngăn xếp trong các cài đặt bằng đệ qui có thể lớn bằng N (chẳng hạn khi tập tin đã được sắp). Điều này rất khó đối với cài đặt bằng đệ qui của Quicksort: luôn luôn tồn tại một ngăn xếp nằm dưới, trường hợp tập tin kích thước lớn có thể làm cho chương trình chấm dứt không bình thường vì thiếu bộ nhớ, tình trạng này dĩ nhiên không được cho phép đối với một chương trình sắp xếp



Hình 9.4 *Các tập tin con trong quicksort (không đệ qui)*

trong thư viện. Dưới đây chúng ta sẽ xem xét cách tạo ra các trường hợp suy thoái nhưng muốn né tránh vấn đề này trong một cài đặt đệ qui rất khó nếu không dùng kỹ thuật khử đệ qui.(Ngay cả chuyển đổi thứ tự xử lý của các tập tin con cũng không giúp được gì hơn.)

Sử dụng thô sơ một ngăn xếp trong chương trình ở trên dẫn đến một chương trình hiệu quả hơn cài đặt đệ qui trực tiếp. Vấn đề là nếu cả hai tập tin con chỉ có một phần tử, một đầu vào với r = l đặt trên ngăn xếp sẽ được lấy ra và hủy ngay. Vậy phải thay đổi chương trình để cho không có những tập tin như vậy trên ngăn



Hình 9.5 *Sơ đồ cây của quá trình phân hoạch trong Quicksort.*

xếp. Thay đổi này hiệu quả hơn khi có một cải tiến kế tiếp được trình bày dưới đây, nó liên quan đến việc lơ đi những tập tin con có kích thước nhỏ.

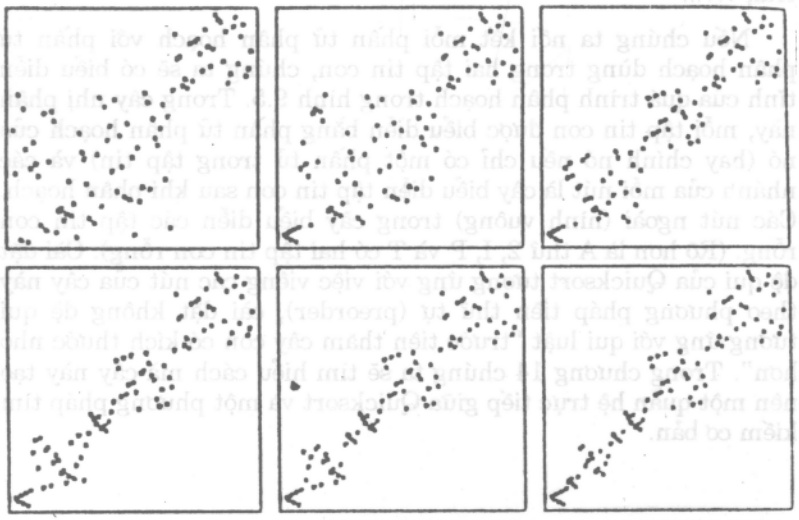
Dĩ nhiên phương pháp không đệ qui xử lý những dạng tập tin con giống như phương pháp có đệ qui với bất kỳ tập tin nào, nó chỉ thực hiện các tập tin con này theo thứ tự khác nhau. Hình 9.4 cho thấy các phân hoạch dùng trong ví dụ đang xét: ba phân hoạch đầu tiên thì giống nhau, nhưng phương pháp không đệ qui phân hoạch tập tin con bên phải của R trước tiên vì nó nhỏ hơn tập tin con bên trái, v.v...

Nếu chúng ta nối kết mỗi phần tử phân hoạch với phần tử phân hoạch dùng trong hai tập tin con, chúng ta sẽ có biểu diễn tĩnh của quá trình phân hoạch trong hình 9.5. Trong cây nhị phân này, mỗi tập tin con được biểu diễn bằng phần tử phân hoạch của nó (hay chính nó nếu chỉ có một phần tử trong tập tin) và các nhánh của mỗi nút là cây biểu diễn tập tin con sau khi phân hoạch. Các nút ngoài (hình vuông) trong cây biểu diễn các tập tin con rỗng. (Rõ hơn là A thứ 2, I, P và T có hai tập tin con rỗng). Cài đặt đệ qui của Quicksort tương ứng với việc viếng các nút của cây này theo phương pháp tiền thứ tự (preorder), cài đặt không đệ qui tương ứng với qui luật "trước tiên thăm cây con có kích thước nhỏ hơn". Trong chương 14 chúng ta sẽ tìm hiểu cách mà cây này tạo nên một quan hệ trực tiếp giữa Quicksort và một phương pháp tìm kiếm cơ bản.

CÁC TẬP TIN CON CÓ KÍCH THƯỚC NHỎ

Cải tiến thứ hai của Quicksort xuất phát từ nhận định là một chương trình đệ qui gọi chính nó trên nhiều tập tin con có kích thước nhỏ vì vậy nên dùng một phương pháp càng hay càng tốt khi gặp các tập tin con có kích thước nhỏ. Hiển nhiên là thay đổi việc kiểm tra ở đầu chương trình đệ qui từ "if r>l" thành lệnh gọi một chương trình sắp xếp bằng chèn; đó là "if r-l<=M then insertion (l,r)". M ở đây là tham số mà giá trị chính xác của nó phụ thuộc vào cài đặt giá trị chọn cho M không cần phải tốt nhất: thuật toán thực hiện giống nhau đối với M trong khoảng từ 5 đến 25. Thời gian chạy rút ngắn 20% đối với đa số các chương trình ứng dụng.

Một cách dễ dàng để xử lý các tập tin con có kích thước nhỏ sao cho hiệu quả hơn là chỉ thay đổi việc kiểm tra ở đầu thành "if r-l>M then", đó là lơ đi những tập tin có kích thước nhỏ trong suốt quá trình phân hoạch. Trong cài đặt không đệ qui, hãy đặt bất kỳ tập tin nào nhỏ hơn M vào ngăn xếp. Sau khi phân hoạch, còn lại một tập tin hầu như được sắp. Như đã đề cập trong chương trước đây, sắp bằng phương pháp chèn là phương pháp được chọn cho những tập tin như vậy, nghĩa là phương pháp này thực hiện



Hình 9.6 *Quicksort (cài đặt đệ qui M = 12)*

đối với tập tin như thế tốt như đối với tập hợp các tập tin có kích thước nhỏ. Phương pháp này nên dè dặt khi sử dụng vì sắp bằng phương pháp chèn thì luôn luôn thực hiện ngay cả khi Quicksort có lỗi làm cho nó không thực hiện gì cả.

Hình 9.6 cho thấy quá trình này trên một mảng thứ tự ngẫu nhiên có kích thước lớn. Đồ hình này mô tả mỗi phân hoạch chia một mảng con thành hai phần độc lập, một mảng con trong hình này được trình bày dưới dạng hình vuông gồm các điểm ngẫu nhiên, quá trình phân hoạch chia hình vuông này thành hai hình vuông nhỏ hơn có một phần tử (phần tử phân hoạch) nằm trên đường chéo. Các phần tử không liên quan đến việc phân hoạch nằm gần với đường chéo, chừa lại một mảng dễ xử lý bằng phương pháp sắp chèn. Đồ hình tương ứng cho cài đặt không đệ qui của Quicksort tương tự, nhưng các phân hoạch được thực hiện theo trình tự khác.

PHẦN TỬ GIỮA CỦA BA PHẦN TỬ PHÂN HOẠCH

Cải tiến thứ ba của Quicksort là dùng phần tử phân hoạch tốt hơn. Khả năng chọn lựa bảo đảm nhất để tránh trường hợp xấu nhất là chọn một phần tử ngẫu nhiên trong mảng làm phần tử phân hoạch. Đây là một ví dụ đơn giản của "thuật toán khả thi" dùng ngẫu nhiên để đạt kết quả tốt bất kể thứ tự nào của dữ liệu nhập. Dữ liệu ngẫu nhiên có thể là một công cụ có ích trong thiết kế thuật toán.

Một cải tiến hữu dụng nữa là lấy ba phần tử trong tập tin, sau đó lấy phần tử giữa của ba phần tử này làm phần tử phân hoạch. Nếu ba phần tử được chọn là phần tử trái, giữa, phải của mảng, tránh dùng các phần tử lính canh: sắp xếp ba phần tử này (dùng ba phép hoán vị như trong chương trước), rồi hoán vị phần tử giữa với a[r-1] và thực hiện thuật toán phân hoạch trên a[l+1..r-2]. Phương pháp cải tiến này được gọi là phương pháp phân hoạch bằng phần tử giữa của ba phần tử.

Phương pháp này giúp cho Quicksort: thứ nhất là làm cho trường hợp xấu nhất không thể xảy ra trong các trường hợp sắp

xếp. Để sắp cần N lần, hai trong ba phần tử đang xét phải nằm trong số các phần tử lớn hay nhỏ nhất trong tập tin, và điều này phải xảy ra trong đa số các phép phân hoạch; thứ hai là nó loại trừ khả năng dùng khóa lính canh để phân hoạch vì chức năng này đã thể hiện bởi ba phần tử đang xét trước khi phân hoạch. Thứ ba nó thực sự rút ngắn tổng thời gian trung bình của thuật toán khoảng 5%.

Liên kết của cài đặt không đệ qui cho phương pháp phần tử giữa để cắt nhỏ tập tin có thể cải tiến thời gian chạy của Quicksort trên cài đặt đệ qui cơ bản khoảng 25% đến 30%. Cải tiến thuật toán có thể dùng phần tử giữa của 5 hay nhiều hơn phần tử nhưng lượng thời gian đạt được rất giới hạn. Nhiều khi phải mã lệnh các vòng lặp bên trong (hay toàn chương trình) bằng ngôn ngữ máy hay hợp ngữ.

CHỌN

Một chương trình ứng dụng liên quan đến sắp xếp không nhất thiết là phải thao tác tìm kiếm phần tử giữa của một tập hợp số. Điều này là phép toán thông thường trong thống kê và các ứng dụng xử lý dữ liệu thay đổi.

Thao tác tìm phần tử giữa là trường hợp đặc biệt của thao tác chọn: tìm phần tử nhỏ nhất thứ k của một tập hợp số. Vì thuật toán không đảm bảo là một phần tử nhỏ nhất thứ k không cần quan tâm k-1 phần tử nhỏ hơn và N-k phần tử lớn hơn, phần lớn các thuật toán chọn có thể trả về k phần tử nhỏ nhất của một tập tin không cần phải tính toán nhiều.

Phương pháp chọn có nhiều ứng dụng trong việc xử lý dữ liệu. Dùng phần tử giữa và các phép thống kê thứ tự khác để chia tập tin thành các nhóm nhỏ hơn rất phổ biến. Thường thì chỉ một phần nhỏ của tập tin lớn giữ lại cho các xử lý thêm; trong trường hợp như vậy, một chương trình có thể chọn 10% đầu tiên của các phần tử trên tập tin có lẽ thích hợp hơn một sắp xếp hoàn toàn.

Chúng ta đã thấy một thuật toán áp dụng phương pháp chọn này rồi. Nếu k khá bé, thì sắp xếp bằng chọn sẽ thực hiện rất tốt, cần thời gian tỷ lệ với Nk; trước tiên tìm phần tử nhỏ nhất, kế tiếp

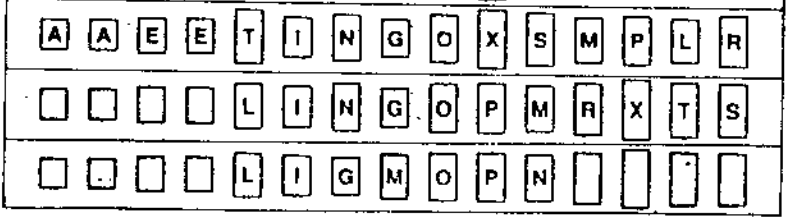
tìm phần tử nhỏ nhất thứ hai bằng cách tìm phần tử nhỏ nhất trong số phần tử còn lại. Đối với k lớn, tìm hiểu phương pháp ở chương 11 sẽ được áp dụng để chạy với thời gian tỷ lệ Nlogk.

Một phương pháp lý thú áp dụng được với các giá trị k và thực hiện thời gian tuyến tính trung bình có thể tính được từ thủ tục phân hoạch dùng cho Quicksort. Nhắc lại là phương pháp phân hoạch của Quicksort để chứa lại một mảng a [1..N] và trả về một số nguyên i sao cho a[1], .. a[i-1] nhỏ hơn hay bằng a[i] và a[i+1] ... a[N] lớn hơn hay bằng a[i]. Nếu tìm thấy phần tử nhỏ nhất thứ k chúng ta có k=i. Mặt khác k<i thì chúng ta phải tìm phần tử nhỏ nhất thứ k ở tập tin con bên trái và nếu ki cần tìm phần tử nhỏ nhất thứ (k-i) ở tập tin con bên phải. Áp dụng lý luận này để tìm phần tử nhỏ nhất thứ k trong mảng a[l..r], đưa đến một thuật toán đệ qui như sau:

```
procedure select(l,r,k:integer);
   var i:integer;
   begin
      if r>l then
      begin i:=partition(l,r);
         if i>l+k-1 then select(l,i-1,k);
         if i<l+k-1 then select(i+1,r,k-i);
      end
   end;
```

Thủ tục này sắp xếp lại mảng sao cho a[l]...a[k-1] nhỏ hơn hay bằng a[h]; và a[k+1] ... a[r] lớn hơn hay bằng a[k].

Ví dụ như lệnh gọi select(1,N,(N+1) div 2) phân hoạch mảng



Hình 9.7 *Phân hoạch để tìm phần tử giữa*

dựa vào các giá trị phần tử giữa. Đối với các khóa trong ví dụ sắp xếp, chương trình này chỉ dùng ba lệnh gọi tìm phần tử giữa như ở hình 9.7. Tập tin được sắp sao cho phần tử giữa đặt vào vị trí mà các phần tử nhỏ hơn sẽ nằm bên trái và lớn hơn nằm bên phải (các phần tử trùng có thể ở một trong hai bên), nhưng nó chưa sắp xong.

Bởi vì thủ tục select luôn kết thúc bằng một lệnh gọi chính nó nên nó thực sự đệ qui trong đó không có ngăn xếp nào dùng để khử đệ qui: khi đến lúc gọi đệ qui chúng ta có thể khởi động lại các tham số và trở về điểm bắt đầu vì không có gì để làm nữa. Chúng ta cũng có thể khử các phép tính đơn giản liên quan đến k như trong cài đặt sau:

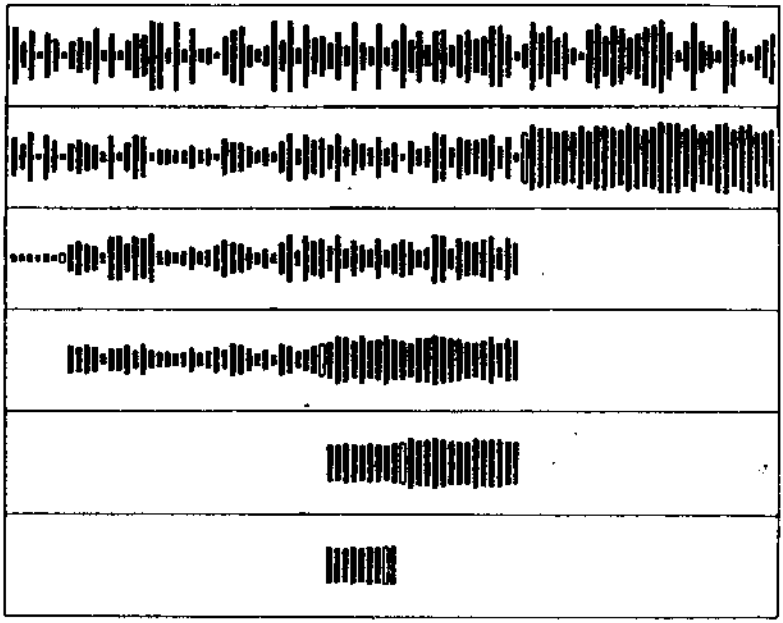
```
procedure select(k:integer);
  var v,t,i,j,l,r:integer;
  begin
  l:=1; r:=N;
  while r>l do
    begin
    v:=a[r]; i:=l-1; j:=r;
    repeat
      repeat i:=i+1 until a[i]>=v;
      repeat j:=j-1 until a[j]<=v;
      t:=a[i]; a[i]:=a[j]; a[j]:=t;
    until j<=i;
    a[j]:=a[i]; a[i]:=a[r]; a[r]:=t;
    if i>=k then r:=i-1;
    if i<=k then l:=i+1;
    end;
  end;
```

Chúng ta sử dụng thủ tục phân hoạch cho Quicksort và có thể thay đổi nó nếu có nhiều khóa trùng nhau.

Hình 9.8 cho thấy quá trình chọn trên tập tin ngẫu nhiên có kích thước lớn. Như với Quicsort chúng ta có thể kết luận là đối với tập tin có kích thước lớn, mỗi phân hoạch nên chia đôi mảng, vì thế toàn bộ quá trình đòi hỏi khoảng

N + N/2 + N/4 + N/8 + ... = 2N so sánh.

TÍNH CHẤT 9.2: *Về mặt trung bình, Phương pháp sắp chọn dựa trên Quicksort có thời gian chạy tuyến tính.*



Hình 9.8 *Tìm phần tử giữa.*

Một phép phân tính tương tự (có phần phức tạp hơn) với phân tích ở trên cho Quicksort dẫn đến kết quả là số so sánh trung bình khoảng $2N + 2k\ln(N/k) + 2(N-k)\ln(N/(N-k))$ tuyến tính với giá trị của k. Đối với k=N/2 (tìm phần tử giữa), ta có khoảng khoảng $(2+2\ln 2)N$ phép so sánh.

Trường hợp xấu nhất giống như đối với Quicksort; dùng phương pháp này để tìm phần tử nhỏ nhất trong tập tin đã sắp rồi đưa đến kết quả thời gian chạy tỉ lệ N^2. Người ta có thể dùng một phần tử tùy ý hay phân hoạch ngẫu nhiên nhưng phải thận trọng: chẳng hạn nếu tìm thấy phần tử nhỏ nhất chúng ta không muốn tập tin chia gần như ở giữa.

Có thể hiệu chỉnh Quicksort dựa vào phương pháp chọn này để bảo đảm thời gian chạy tuyến tính. Sửa đổi này có thể cần thiết về mặt lý thuyết nhưng cực kỳ phức tạp và không thực tế cho lắm.

BÀI TẬP

1. Cài đặt một thuật toán Quicksort đệ qui với sự cắt xén bớt phép sắp chèn cho các tập tin con có ít hơn M phần tử và xác định theo kinh nghiệm giá trị của M mà nó sẽ chạy nhanh nhất trên một tập tin ngẫu nhiên có 1000 phần tử.

2. Giải bài toàn trên đối với một bản cài đặt không đệ qui.

3. Giải bài toán trên có bổ sung phép chọn phần tử giữa của ba phần tử.

4. Quicksort sẽ thực hiện khoảng bao lâu để sắp một tập tin gồm N phần tử bằng nhau.

5. Số lần tối đa mà phần tử lớn nhất có thể được di chuyển trong lúc thi hành Quicksort là bao nhiêu?

6. Hãy chỉ ra làm thế nào tập tin A B A B A B A được phân hoạch, sử dụng hai phương pháp đã đề nghị trong tài liệu.

7. Quicksort dùng bao nhiêu phép so sánh để sắp các khóa E A S Y Q U E S T I O N ?

8. Cần bao nhiêu khóa "cầm canh" nếu phương pháp sắp chèn được gọi một cách trực tiếp từ trong Quicksort ?

9. Có hợp lý không khi dùng một hàng đợi thay vì một ngăn xếp cho một bản cài đặt không đệ qui của Quicksort ? Tại sao có và tại sao không ?

10. Viết một chương trình để tổ chức lại một tập tin sao cho tất cả các phần tử với các khóa bằng với giá trị trung bình thì nằm tại chỗ, với các phần tử nhỏ hơn thì nằm bên trái và các phần tử lớn hơn thì nằm bên mặt.

10
SẮP XẾP BẰNG CƠ SỐ

Các "khóa" dùng để xác định thứ tự của các mẩu tin của tập tin đối với nhiều chương trình ứng dụng sắp xếp rất phức tạp (chẳng hạn thứ tự dùng trong sổ điện thoại hay một thư mục sách). Vì lý do nầy, để xác định các phương pháp sắp xếp có hai thao tác cơ bản rất hợp lý là "so sánh hai khóa" và "hoán vị hai mẩu tin". Đa số các phương pháp chúng ta đã nghiên cứu có thể mô tả dựa trên hai thao tác cơ bản. Tuy nhiên đối với nhiều chương trình ứng dụng, có thể lợi dụng sự kiện là các khóa có thể là số nằm trong một dãy hữu hạn nào đó. Các phương pháp sắp xếp lợi dụng tính chất số được gọi là **sắp xếp dựa vào cơ số** (radix sort). Các phương pháp nầy không chỉ so sánh khóa: chúng xử lý và so sánh các phần của khóa.

Thuật toán sắp xếp theo cơ số xem các khóa như là số được biểu diễn ở dạng hệ cơ số M đối với các giá trị khác nhau của M và làm việc với các ký số riêng lẻ. Chẳng hạn, một thư ký phải sắp xếp một xấp thẻ có in ba chữ số trên thẻ. Một cách thực hiện hợp lý đối với anh ta là tạo ra 10 chồng: một cho các số 100, một cho các số giữa 100 và 199 v.v..., đặt các thẻ vào trong chồng, sau đó xử lý các chồng riêng biệt, bằng cách sử dụng cùng phương pháp trên các số kế tiếp hay dùng phương pháp dễ hơn nếu chỉ có một vài thẻ. Đây là một ví dụ đơn giản của sắp xếp cơ số với M=10. Chúng ta sẽ xem xét chi tiết phương pháp nầy và vài phương pháp khác trong chương nầy. Dĩ nhiên với phần lớn các máy tính thì rất thuận lợi để làm việc với M=2 (hay lũy thừa của 2) hơn là M=10. Mọi biểu diễn bên trong máy tính đều có dạng nhị phân, vì nhiều ứng dụng có sắp xếp đều chuyển đều chuyển thành sắp xếp bằng cơ số thao tác trên khóa là số nhị phân. Nhưng tiếc là Pascal và nhiều ngôn ngữ khác gây nhiều khó khăn khi viết chương trình phụ thuộc vào biểu diễn số nhị phân. (Lý do là Pascal chỉ là một ngôn

ngữ diễn đạt chương trình độc lập với ngôn ngữ máy và các máy tính khác nhau có thể sử dụng các biểu diễn khác nhau cho cùng một dạng số).

Ngôn ngữ Pascal loại bỏ nhiều kiểu kỹ thuật trên bit, và xử lý tốt hơn cho các cấu trúc như mẩu tin và tập hợp; tuy nhiên sắp xếp bằng cơ số dường như là một trường hợp ngẫu nhiên ngược lại với giả thiết của ngôn ngữ Pascal. May là không khó khi dùng các phép toán số học để mô phỏng các thao tác cần dùng, vì vậy chúng ta có thể viết các chương trình Pascal để mô tả thuật toán mà có thể biên dịch dễ dàng thành các chương trình hiệu quả viết bằng ngôn ngữ lập trình cung cấp các thao tác bit trên số nhị phân.

BIT

Cho trước một khóa biểu diễn dưới dạng số nhị phân, thao tác cơ bản cần cho phương pháp sắp xếp dựa vào cơ số là trích tập hợp các bit kề nhau của một số. Giả sử chúng ta xử lý các khóa là các số nguyên giữa 0 và 1000. Các số nầy được biểu diễn bằng số nhị phân chiếm 10 bit. Trong ngôn ngữ máy các bit được trích ra từ số nhị phân bằng cách dùng phép toán trên bit là "and" và "shift". Chẳng hạn hai bit đầu của số 10 được trích ra bằng cách đẩy sang phải 8 vị trí bit sau đó dùng phép and với mặt nạ bit 0000000011. (Trong Pascal các thao tác nầy có thể được mô phỏng bằng div và mod.) Ví dụ như hai bit đầu của một số x 10 bit có thể lấy được bằng phép tính (x div 256) mod 4. Nói chung "đẩy x sang phải k vị trí bit có thể mô phỏng bằng phép tính x div 2^k" và "làm cho tất cả các bit bằng 0 ngoại trừ j bit bên phải nhất của x" có thể mô phỏng bằng phép tính x mod 2^j. Trong mô tả của thuật toán sắp cần phải có một hàm bits(x,k,j:integer):integer liên kết các thao tác nầy để trả về j bit, trong j bit nầy xuất hiện k bit phải của x bằng cách tính (x div 2^k) mod 2^j. Ví dụ, bit phải nhất của x được lấy ra từ lệnh gọi bits(x,0,1). Hàm nầy có thể làm hiệu quả hơn bằng cách tính trước (hay định nghĩa hằng) các lũy thừa của 2. Chú ý là một chương trình chỉ sử dụng hàm nầy sẽ sắp xếp dựa vào cơ số mặc dù chúng ta có thể hy vọng hiệu quả hơn nếu biểu diễn ở dạng nhị phân và trình biên dịch khá tốt để có thể tính toán bằng lệnh ngôn ngữ máy "shift" và "and". Nhiều cài đặt bằng Pascal có

phần mở rộng ngôn ngữ cho phép các thao tác nầy được làm một cách trực tiếp.

Chúng ta xem xét hai kiểu sắp xếp dựa vào cơ số mà thứ tự xem xét các bit của khóa thì khác nhau. Chúng ta giả định rằng các khóa không ngắn, cho nên rất cần trích ra các bit của chúng. Nếu các khóa ngắn thì sử dụng phương pháp đếm quá trình phân phối trong chương 8. Nhắc lại là phương pháp nầy có thể sắp N khóa là số nguyên giữa 0 và M-1 theo thời gian tuyến tính, sử dụng một mảng phụ có kích thước M để đếm và một mảng khác kích thước N để sắp xếp lại mẩu tin. Nếu chúng ta có thể dùng một mảng kích thước 2^b thì có thể sắp xếp dễ dàng các khóa b-bit theo thời gian tuyến tính. Nếu các khóa khá dài (b=32) phương pháp nầy có thể không thực hiện được.

Phương pháp cơ bản đầu tiên để sắp xếp bằng cơ số là kiểm tra các bit trong các khóa từ trái sang phải. Nó dựa trên tính chất là kết quả của "so sánh" khóa giữa hai khóa phụ thuộc vào giá trị của các bit đầu. Vì vậy, tất cả các khóa có bit bắt đầu bằng 0 xuất hiện trước tất cả các khóa có bit bắt đầu là 1, khóa nào có bit thứ hai là 0 sẽ xuất hiện trước các khóa có bit thứ hai là 1, và cứ như vậy. Phương pháp sắp xếp bằng cơ số từ trái sang phải gọi là **sắp xếp hoán vị cơ số**, là phương pháp sắp xếp bằng cách chia các khóa một cách có hệ thống.

Phương pháp cơ bản thứ hai mà chúng ta sẽ xem xét được gọi là **sắp xếp cơ số trực tiếp**, kiểm tra các bit trong khóa từ phải qua trái. Nó dựa vào một nguyên tắc là rút ngắn việc sắp xếp trên khóa b-bit thành b lần sắp xếp trên khóa 1 bit.

SẮP XẾP HOÁN VỊ CƠ SỐ

Giả sử chúng ta có thể sắp xếp các mẩu tin của tập tin sao cho tất cả các mẩu tin có khóa bắt đầu bằng bit 0 đứng trước các mẩu tin có khóa bắt đầu bằng bit 1. Điều nầy dẫn đến một phương pháp sắp xếp đệ qui như sau: nếu hai tập tin con được sắp xếp độc lập, thì toàn bộ tập tin sẽ được sắp. Sự sắp xếp lại của tập tin rất giống như trong phân hoạch của Quicksort: quét từ trái để tìm khóa bắt đầu bằng bit có giá trị 1, quét từ phải để tìm khóa bắt đầu bằng bit

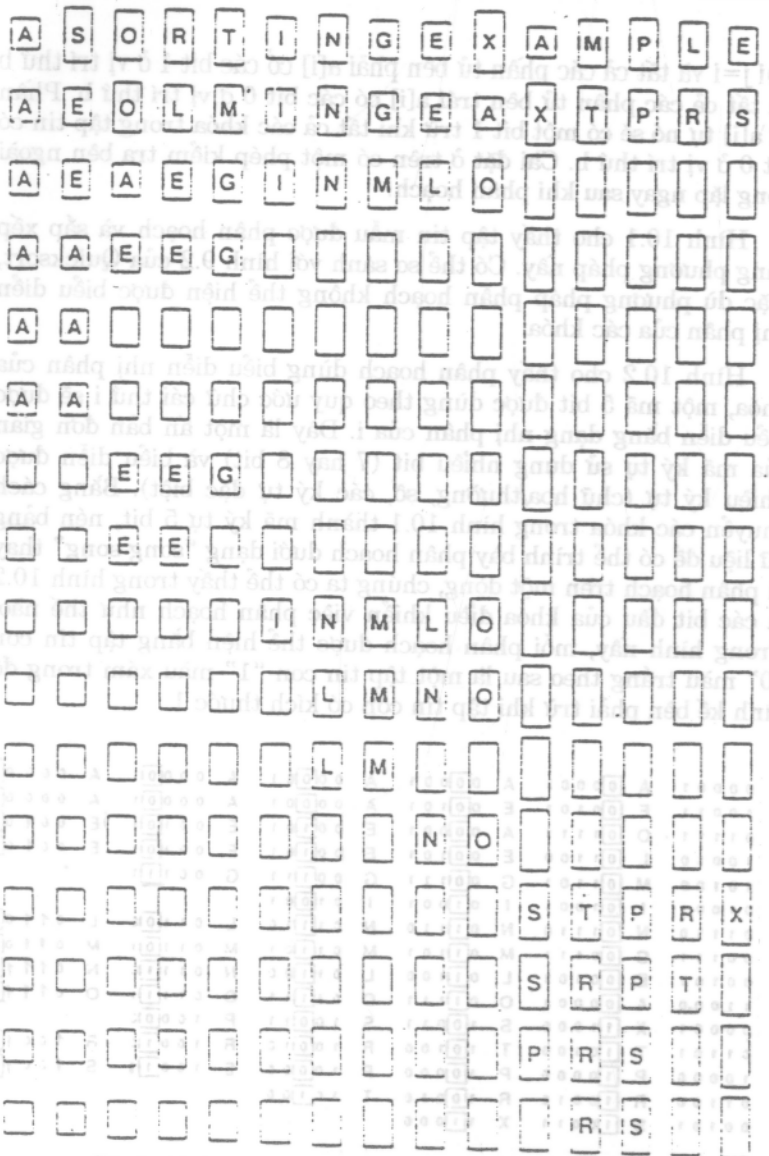
có giá trị 0, hoán vị và tiếp tục quá trình cho đến khi các con trỏ quét giao nhau.

Điều nầy dẫn đến một thủ tục sắp xếp đệ qui tương tự với Quicksort như sau:

```
procedure radixexchange(l,r,b:integer);
   var t,i,j: integer;
   begin
   if (r>l) and (b>=0) then
      begin
      i:=l; j:=r;
      repeat
         while (bits(a[i],b,1)=0) and (i<j) do i:=i+1;
         while (bits(a[j],b,1)=0) and (i<j) do j:=j-1;
      until j=i;
      if bits(a[r],b,1)=0 then j:=j+1; radixexchange(l,j-1,b-1);
      radixexchange(j,r,b-1);
      end;
   end;
```

Để đơn giản, giả sử là a[1..N] chứa các số nguyên dương nhỏ hơn 2^{32} (sao cho chúng có thể biểu diễn như số nhị phân 31 bit), kế đó thủ tục radixexchange(1,N,30) sẽ sắp xếp mảng. Biến b giữ lại dấu tích của các bit đã kiểm tra, miền giá trị từ 30 (trái nhất) xuống đến 0 (phải nhất). Thông thường có thể áp dụng cài đặt các bit cho biểu diễn trong máy cho số âm sao cho số âm có thể được xử lý khi sắp xếp.

Cài đặt nầy dĩ nhiên tương tự với cài đặt đệ qui của Quicksort trong chương 9. Chủ yếu việc phân hoạch trong sắp xếp hoán vị cơ số giống như phân hoạch của Quicksort ngoại trừ số 2^b được dùng như phần tử phânhoạch thay vì dùng số khác trong tập tin. Bởi vì 2^b có thể không chứa trong tập tin nên không bảo đảm là một phần tử được đặt vào vị trí cuối cùng của nó trong suốt quá trình phân hoạch. Bởi vì chỉ duy nhất một bit được kiểm tra, nên chúng ta có thể dựa vào các phần tử lính canh để ngưng việc quét: vì vậy việc kiểm tra (i<j) cần phải có trong vòng lặp quét. Đối với Quicksort, một phép hoán vị nữa được thực hiện trong trường hợp j=i, nhưng không cần trở lại phép hoán vị nầy bên ngoài vòng lặp vì "phép hoán vị" là của a[i] với chính nó. Việc phân hoạch ngừng

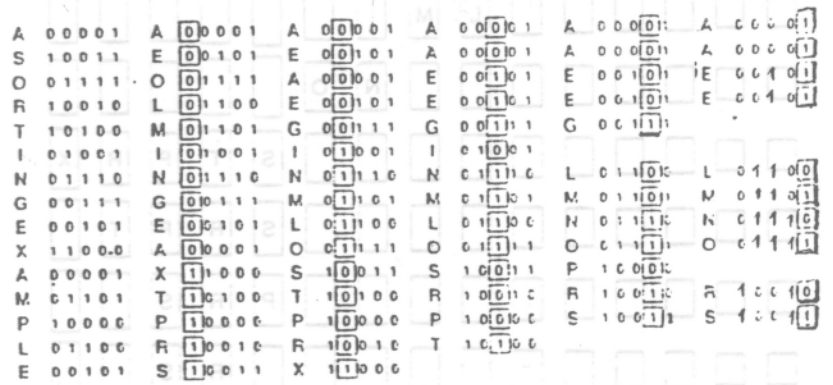


Hình 10.1 *Các tập tin con trong sắp xếp hoán vị cơ số*

khi j=i và tất cả các phần tử bên phải a[i] có các bit 1 ở vị trí thứ b và tất cả các phần tử bên trái a[i] có các bit 0 ở vị trí thứ b. Phần tử a[i] tự nó sẽ có một bit 1 trừ khi tất cả các khóa trong tập tin có bit 0 ở vị trí thứ b. Cài đặt ở trên có một phép kiểm tra bên ngoài vòng lặp ngay sau khi phân hoạch.

Hình 10.1 cho thấy tập tin mẫu được phân hoạch và sắp xếp bằng phương pháp nầy. Có thể so sánh với hình 9.2 của Quicksort, mặc dù phương pháp phân hoạch không thể hiện được biểu diễn nhị phân của các khóa.

Hình 10.2 cho thấy phân hoạch dùng biểu diễn nhị phân của khóa, một mã 5 bit được dùng theo quy ước chữ cái thứ i sẽ được biểu diễn bằng dạng nhị phân của i. Đây là một ấn bản đơn giản của mã ký tự sử dụng nhiều bit (7 hay 8 bit) và biểu diễn được nhiều ký tự (chữ hoa/thường, số, các ký tự đặc biệt). Bằng cách chuyển các khóa trong hình 10.1 thành mã ký tự 5 bit, nén bảng dữ liệu để có thể trình bày phân hoạch dưới dạng "song song" thay vì phân hoạch trên một dòng, chúng ta có thể thấy trong hình 10.2 là các bit đầu của khóa điều khiển việc phân hoạch như thế nào. Trong hình nầy, mỗi phân hoạch được thể hiện bằng tập tin con "0" màu trắng theo sau là một tập tin con "1" màu xám trong đồ hình kế bên phải trừ khi tập tin con có kích thước 1.

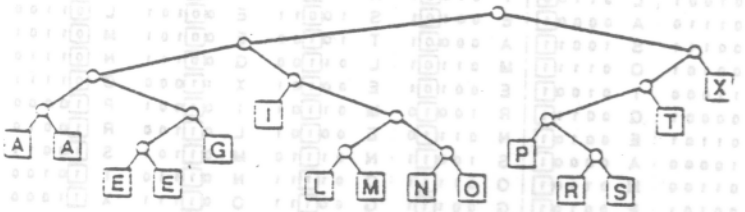


Hình 10.2 *Sắp xếp hoán chuyển cơ số (sắp xếp cơ số "trái sang phải")*

Một vấn đề quan trọng của sắp xếp bằng cơ số không thể hiện trong ví dụ này là các phép phân hoạch bị suy thoái (phân hoạch có các khóa cùng giá trị trong bit được dùng) có thể xảy ra thường xuyên. Tình huống này xảy ra thông thường trong các tập tin khi sắp xếp các số nhỏ (có nhiều bit 0 ở đầu). Điều này cũng xảy ra đối với các ký tự: chẳng hạn giả sử các khóa 32 bit dùng cho 4 ký tự bằng cách mã hóa mỗi ký tự trong một mã 8 bit chuẩn và nối chúng lại với nhau. Các phân hoạch bị suy thoái có thể xảy ra ở vị trí bắt đầu của mỗi ký tự khi các chữ cái bắt đầu bằng các bit giống nhau trong phần lớn các mã ký tự. Nhiều hậu quả tương tự khác đáng chú ý khi dữ liệu bị mã hóa.

Hình 10.2 cho thấy khi một khóa được phân biệt với các khóa khác nhờ các bit trái của nó không có thêm bit nào được kiểm tra. Điều này tiện lợi trong một số tình huống, nhưng lại bất tiện trong một số tình huống khác. Khi các khóa thực sự là các bit ngẫu nhiên, mỗi khóa khác với khóa sau khoảng lgN bit, số bit này có thể ít hơn số bit trong khóa. Trong trường hợp ngẫu nhiên chúng ta mong muốn mỗi phân hoạch chia đôi tập tin con. Chẳng hạn, sắp một tập tin con có 1000 mẩu tin có thể liên quan đến việc kiểm tra khoảng 10 hay 11 bit trên mỗi khóa (mặc dù khóa là khóa 32 bit). Mặt khác, chú ý là tất cả các bit trong khóa trùng đều được kiểm tra. Sắp xếp dựa vào cơ số không hoạt động tốt trên các tập tin chứa nhiều khóa giống nhau. Sắp xếp hoán vị cơ số thực sự nhanh hơn Quicksort nếu các khóa tham gia sắp xếp gồm có các bit ngẫu nhiên, nhưng Quicksort có thể áp dụng tốt hơn trong các trường hợp ít ngẫu nhiên.

Hình 10.3 là cây biểu diễn quá trình phân hoạch của sắp xếp



Hình 10.3 *Sơ đồ cây của quá trình phân hoạch trong sắp xếp hoán vị cơ số*

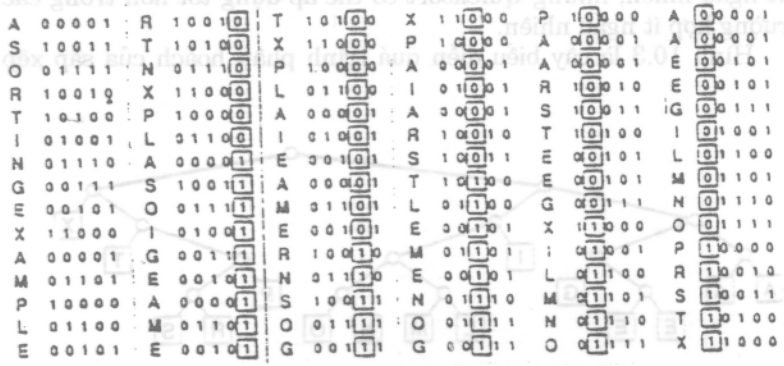
hoán vị cơ số có thể so sánh với hình 9.5. Trong cây nhị phân nầy các nút trong biểu diễn cho các điểm phân hoạch, các nút ngoài là các khóa trong tập tin, tất cả các nút sẽ nằm trong tập tin có kích thước 1. Trong chương 17 chúng ta sẽ thấy cây nầy thể hiện mối quan hệ trực tiếp giữa sắp xếp hoán vị cơ số và một phương pháp sắp xếp cơ bản.

Cài đặt đệ qui ở trên có thể được cải tiến bằng cách khử đệ qui và sử dụng các tập tin con có kích thước nhỏ như đã làm trong Quicksort.

SẮP XẾP CƠ SỐ TRỰC TIẾP

Một phương pháp sắp xếp dựa vào cơ số là dựa vào việc kiểm tra các bit từ phải sang trái, đây là phương pháp do máy tính cũ sử dụng. Hình 10.4 cho thấy một phương pháp sắp xếp dựa vào cơ số theo cách từ phải sang trái, hết bit nầy đến bit khác thực hiện trên các khóa mẫu. Cột thứ i trong hình 10.4 được sắp xếp trên giải bit thứ i của khóa và có được từ cột thứ i-1 bằng cách trích ra tất cả các khóa có bit thứ i là 0, kế đó là các khóa có bit thứ i là 1.

Không dễ thuyết phục rằng phương pháp nầy hoạt động, thực sự nó không làm gì cả trừ khi quá trình phân hoạch một bit ổn định. Khi sự ổn định được thỏa mãn, ta có thể chứng minh ngay phương pháp nầy hoạt động tốt nếu chú ý: sau khi đặt các khóa có



Hình 10.4 *Sắp xếp cơ số trực tiếp*

bit thứ i là 0 thì sau đó là các khóa có bit thứ i là 1. Chúng ta biết là hai khóa bất kỳ xuất hiện theo thứ tự phù hợp trong tập tin nếu bit thứ i của chúng khác nhau, hay nếu bit thứ i của chúng bằng nhau thì thứ tự cũng phù hợp nhờ vào tính ổn định. Yêu cầu của tính ổn định là phương pháp phân hoạch được dùng trong sắp xếp hoán vị cơ số ứng dụng cho sắp xếp tự phải qua trái.

Phân hoạch giống như sắp xếp tập tin chỉ có hai giá trị và sắp xếp bằng cách đếm sự phân phối ở chương 8 rất thích hợp cho điều kiện nầy. Nếu chúng ta giả định là M=2 trong chương trình đếm sự phân phối và thay thế a[i] bằng bits(a[i],k,1), giả sử rằng chương trình trở thành một phương pháp để sắp xếp các phần tử của mảng a trên bit k từ phía phải và đặt kết quả vào mảng tạm thời t. Nhưng không có lý do nào để dùng M=2, chúng ta nên cho M càng lớn càng tốt, và thấy là cần một mảng chứa M lần đếm. Điều nầy tương ứng với việc sử dụng m bit trong suốt quá trình sắp xếp với $M=2^m$. Phương pháp sắp xếp cơ số trực tiếp, xét về mặt nào đó thì là dạng tổng hóa của của sắp xếp bằng phương pháp đếm phân phối, xem cài đặt sau đây cho việc sắp a[1..N] trên w bit bên phải nhất:

```
procedure straightradix;
   var i,j,pass: integer;
      count: array[0..M] of integer;
   begin
   for pass:=0 to (w div m)-1 do
      begin
      for j:=0 to M-1 do count[j]:=0;
      for i:=1 to N do
         count[bits(a[i],pass*m,m)]:=count[bits(a[i],pass*m,m)]+1;
      for j:=1 to M-1 do count[j]:= count[j-1]+count[j];
      for i:=N downto 1 do
         begin
         b[count[bits(a[i], pass*m,m)]]:=a[i];
         count[bits(a[i], pass*m,m)]:=count[bits(a[i],pass*m,m)]-1;
         end;
      for i:=1 to N do a[i]:=b[i]
      end;
   end;
```

Thủ tục nầy sử dụng hai lệnh gọi thủ tục bits để tăng và giảm biến đếm count. Đẳng thức $M=2^m$ được lưu giữ trong các tên biến, mặc dù một vài ấn bản của Pascal không phân biệt được m và M.

Thủ tục trên chỉ thực hiện tốt khi w là bội số của m. Thông thường đây không phải là một giả thiết giới hạn của sắp xếp dựa vào cơ số: nó tương ứng với việc chia các khóa thành một số mẩu có kích thước bằng nhau.

Phương pháp cài đặt trên chuyển tập tin a sang b trong từng lần phân phối, rồi chuyển lại a bằng một vòng lặp đơn giản. Vòng lặp "sao chép mảng" có thể bỏ qua nếu muốn, bằng cách tạo hai bảng sao chép của đoạn chương trình đếm việc phân phối, một bản để sao chép từ a qua b, bản kia để sắp từ b qua a.

HIỆU QUẢ CỦA PHƯƠNG PHÁP SẮP XẾP BẰNG CƠ SỐ

Thời gian chạy của cả hai phương pháp sắp xếp dựa vào cơ số nói trên khi sắp N mẩu tin b-bit là Nb. Người ta cũng có thể nghĩ rằng thời gian chạy tương đương NlogN, vì nếu tất cả các số khóa đều khác nhau thì b ít nhất phải bằng logN. Mặt khác, cả hai phương pháp luôn dùng ít hơn Nb thao tác: thứ nhất là phương pháp từ-trái-qua-phải (vì nó có thể dừng khi một khi thấy sự khác nhau giữa các khóa), thứ hai phương pháp từ-phải-qua-trái (vì nó xử lý nhiều bit cùng một lúc).

TÍNH CHẤT 10.1 *Phương pháp sắp xếp dựa vào cơ số dùng trung bình khoảng NlgN phép so sánh bit.*

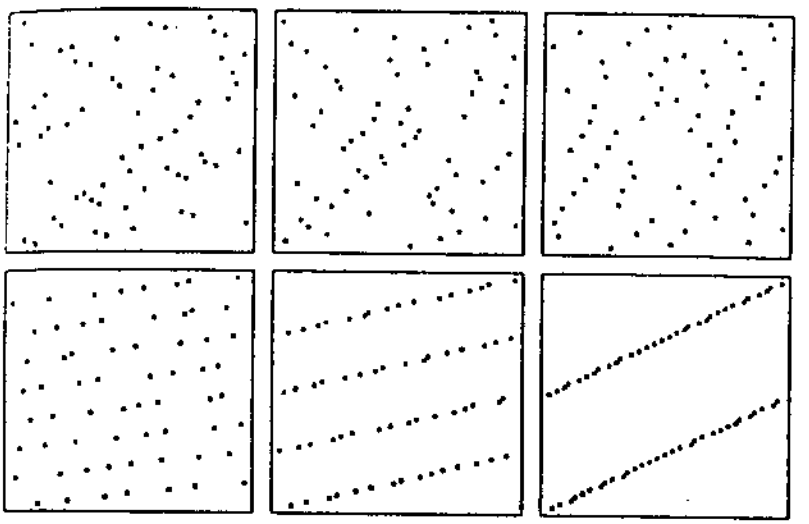
Nếu kích thước tập tin là lũy thừa của 2 và các bit phân bố ngẫu nhiên, chúng ta muốn một nửa các bit đầu bằng 0 và nửa còn lại bằng 1, vì thế công thức truy hồi $C_N=2C_{N/2}+N$ sẽ mô tả tính năng giống như Quicksort ở chương 9. Mô tả của trường hợp nầy cũng không chính xác vì phân hoạch chỉ rơi vào giữa trong trường hợp trung bình (và vì số bit trong khóa là hữu hạn). Tuy nhiên, đối với mô hình nầy, phân hoạch có lẻ rơi vào phần giữa nhiều hơn Quicksort, vì vậy kết quả được khẳng định là đúng. (Việc chứng minh khẳng định nầy vượt khỏi phạm vi của chúng ta).

TÍNH CHẤT 10.2 *Cả hai phương pháp sắp xếp bằng cơ số cần ít hơn Nb phép so sánh bit khi sắp xếp N khóa b-bit.*

Phương pháp sắp xếp dựa vào cơ số có thời gian tuyến tính theo nghĩa là thời gian sử dụng tỉ lệ với số bit của dữ liệu nhập. Điều này có thể quan sát trực tiếp từ việc kiểm chứng chương trình: không có bit nào được kiểm tra nhiều hơn một lần.

Đối với các tập tin lớn, ngẫu nhiên, phương pháp sắp xếp dựa vào cơ số trực tiếp biểu hiện khác hơn. Hình 10.5 cho thấy các giai đoạn của sắp xếp cơ số trực tiếp trên một tập tin ngẫu nhiên với các khóa 5-bit. Diễn tiến của tổ chức tập tin trong quá trình sắp xếp được trình bày rõ trong đồ hình này. Ví dụ như sau giai đoạn thứ 3 (tận cùng bên trái), tập tin chứa bốn tập tin con được sắp lẫn lộn: các khóa bắt đầu bằng 00, các khóa bắt đầu bằng 01, và cứ như thế.

TÍNH CHẤT 10.3 *Phương pháp sắp xếp cơ số trực tiếp có thể sắp N mẩu tin với các khóa b-bit trong b/m lần lặp, cần chỗ trống cho 2^m biến đếm (và một bộ đệm cho việc sắp xếp các tập tin).*



Hình 10.5 *Các giai đoạn của sắp xếp bằng cơ số trực tiếp.*

Để chứng minh tính chất này có thể đi thẳng vào phần cài đặt. Nếu có thể lấy m=b/4 mà không cần thêm bộ nhớ, chúng ta sẽ có phương pháp sắp xếp tuyến tính. Vấn đề thực tế của tính chất này sẽ được trình bày chi tiết hơn trong phần sau.

Xem hình 10.5 Các giai đoạn của sắp xếp bằng cơ số trực tiếp.

PHƯƠNG PHÁP SẮP XẾP TUYẾN TÍNH

Cài đặt của phương pháp sắp xếp bằng cơ số trực tiếp đã nêu ở phần trước tạo b/m lần lặp trên tập tin. Cho m lớn, ta có một phương pháp sắp xếp rất hiệu quả miễn là ta có sẵn $M=2^m$ ô bộ nhớ. Một cách chọn hợp lý là cho m bằng 1/4 kích thước ô nhớ (nghĩa là b/4), khi đó phương pháp sắp xếp qua bốn lần đếm phân phối. Các khóa được sử dụng như cơ số M, và mỗi ký số của khóa được kiểm tra, nhưng chỉ có 4 ký số trên một khóa (điều này tương thích với cấu trúc máy tính có kích thước ô nhớ 32 bit, là 4 byte mỗi byte 8 bit). Mỗi lần lặp bước đếm phân phối thì tuyến tính, và vì chỉ có 4 byte nên toàn bộ sắp xếp tuyến tính, đây là đặc trưng tốt nhất mà chúng ta mong đợi cho một phương pháp sắp xếp.

Thật ra chỉ cần hai lần đếm phân phối, chúng ta có thể sắp xếp bằng cách lợi dụng tính chất là tập tin hầu như được sắp mà chỉ cần dùng b/2 bit đầu của khóa b-bit. Giống như Quicksort, việc sắp xếp sẽ hiệu quả hơn nếu dùng sắp xếp bằng phương pháp chèn trên toàn bộ các tập tin sau đó. Phương pháp này dĩ nhiên là hiệu chỉnh cho cài đặt ở trên: để thực hiện một phương pháp sắp từ phải qua trái sử dụng một nửa đầu của khóa, chúng ta chỉ cần bắt đầu vòng lặp ngoài pass=b div (2*m) thay vì pass=1. Kế đến là dùng phương pháp sắp xếp chèn trên tập tin gần như đã sắp thứ tự. Để kiểm chứng tập tin được sắp xếp tốt nếu nó đã được xếp theo thứ tự của các bit dẫn đầu, độc giả có thể kiểm chứng cột đầu tiên của hình 10.2. Chẳng hạn, nếu phương pháp sắp xếp chèn áp dụng cho tập tin đã xếp rồi trên 3 bit đầu tiên thì ta chỉ cần sáu phép hoán vị.

Đối với các tập tin có kích thước lớn và các khóa có các bit ngẫu nhiên, ta sử dụng hai lần phép đếm phân phối (m bằng khoảng 1/4 kích thước ô nhớ) và dùng phương pháp sắp xếp chèn

thì hầu như nó chạy nhanh hơn bất kỳ phương pháp nào khác. Bất tiện của nó là cần thêm một mảng cùng kích thước với mảng cần sắp. Có thể bỏ mảng nầy đi và dùng kỹ thuật xâu liên kết, nhưng không gian lưu trữ cần thiết vẫn tỉ lệ với N.

Phương pháp sắp xếp tuyến tính dĩ nhiên rất hữu dụng trong các trình ứng dụng, nhưng có nhiều lý tại sao nó không được dùng phổ biến. Trước tiên vì tính năng của nó phụ thuộc vào khóa có các bit ngẫu nhiên với thứ tự sắp ngẫu nhiên. Nếu không thỏa điều kiện nầy, quá trình thực hiện sẽ bị giảm hiệu quả. Thứ hai là phương pháp nầy cần chổ lưu trữ tỉ lệ với kích thước mảng cần sắp. Thứ ba, "vòng lặp trong" của chương trình thực sự chứa chỉ một vài lệnh, vì thế dù cho phương pháp nầy tuyến tính nó cũng không nhanh hơn Quicksort cho lắm, ngoại trừ trường hợp tập tin có kích thước lớn. Chọn lựa giữa Quicksort và sắp xếp dựa vào cơ số rất khó, nó không chỉ phụ thuộc vào đặc trưng của chương trình ứng dụng (như khóa, mẩu tin, kích thước tập tin) mà còn phụ thuộc vào đặc trưng của môi trường thiết bị và lập trình mà liên quan đến tính hiệu quả truy xuất và sử dụng các bit riêng lẻ.

BÀI TẬP

1. So sánh số hoán vị được dùng bởi sắp xếp hoán vị cơ số với số hoán vị được dùng bởi Quicksort cho tập tin gồm các khóa 001, 011, 101, 110, 000, 001, 010, 111, 110, 010.

2. Tại sao lại không quan trọng khi khử đệ qui từ phương pháp sắp xếp hoán vị cơ số như là đối với Quicksort.

3. Hãy sửa đổi phương pháp sắp xếp hoán vị cơ số để nhảy qua các bit dẫn đầu giống nhau trên tất cả các khóa. Trong những trường hợp nào thì điều nầy trở nên phung phí vô ích?

4. Điều sau đây đúng hay sai: thời gian thực hiện của phương pháp sắp xếp cơ số trực tiếp không phụ thuộc vào thứ tự các khóa trong tập tin nhập. Hãy giải thích câu trả lời của bạn.

5. Phương pháp nào sẽ nhanh hơn đối với tập tin gồm các khóa bằng nhau: sắp xếp hoán vị cơ số hay sắp xếp cơ số trực tiếp?

6. Điều sau đây đúng hay sai: cả hai phương pháp sắp xếp hoán vị cơ số và sắp xếp cơ số trực tiếp sẽ kiểm tra tất cả các bit của tất cả các khóa trong tập tin. Hãy giải thích câu trả lời của bạn.

7. Trừ ra yêu cầu về bộ nhớ bổ sung, hãy cho biết bất tiện chính của chiến lược thực hiện phép sắp cơ số trực tiếp trên các bit đi đầu của các khóa, rồi tiếp tục bằng phép sắp chèn sau đó.

8. Cần dùng bao nhiêu bộ nhớ để thực hiện một phép sắp cơ số trực tiếp 4-đường của N khóa b-bit?

9. Kiểu nào của tập tin nhập sẽ khiến cho phép sắp hoán vị cơ số chạy chậm nhất (đối với một N rất lớn)?

10. Hãy so sánh theo kinh nghiệm phép sắp cơ số trực tiếp với phép sắp hoán vị cơ số đối với một tập tin gồm 1000 khóa 32-bit?

11
HÀNG ĐỢI CÓ ĐỘ ƯU TIÊN

Trong nhiều chương trình ứng dụng, các mẩu tin có khóa phải được xử lý theo thứ tự, nhưng không cần thiết theo thứ tự được sắp hoàn chỉnh. Thông thường tập hợp mẩu tin phải được tập trung lại rồi xử lý mẩu tin có kích thước lớn nhất, lại tập trung tiếp nhiều mẩu tin và xử lý trên mẩu tin lớn nhất kế tiếp và cứ tiếp tục như vậy. Một cấu trúc dữ liệu thích hợp trong môi trường như thế hỗ trợ cho thao tác chèn một phần tử mới và hủy phần tử lớn nhất. Một cấu trúc ngược lại với hàng đợi (hủy phần tử cũ nhất) và ngăn xếp (hủy phần tử mới nhất) được gọi là hàng đợi có độ ưu tiên, sử dụng các phép gán độ ưu tiên thích hợp.

Các ứng dụng của hàng đợi có độ ưu tiên bao gồm các hệ thống mô phỏng (các khóa tương ứng với "thời gian của biến cố" phải được xử lý theo trình tự); lịch thực hiện trong hệ máy tính (các khóa tương ứng với "độ ưu tiên" cho biết người dùng nào sẽ được xử lý trước) và các phép tính số học (khóa có thể là lỗi tính toán nên khóa lớn nhất sẽ được thực hiện trước tiên).

Chúng ta sẽ xem cách sử dụng hàng đợi có độ ưu tiên là nền tảng cho các thuật toán cải tiến ở phần sau. Trong chương 22, chúng ta sẽ phát triển một thuật toán nén tập tin có dùng các chương trình trong chương này và trong chương 31 và 33, chúng ta sẽ tìm hiểu xem các hàng đợi có thể ứng dụng một cách cơ bản cho vài thuật toán tìm kiếm. Cũng có vài ví dụ hàng đợi đóng vai trò quan trọng như là một công cụ trong thiết kế hệ thống.

Cách thao tác trên hàng đợi có độ ưu tiên thì chính xác hơn vì có vài thao tác mà chúng ta cần thực hiện trên hàng đợi để bảo quản chúng và sử dụng chúng một cách có hiệu quả cho các ứng dụng như đã đề cập ở trên. Thực ra lý do chính mà hàng đợi rất có ích là tính linh động cho phép các thao tác khác nhau thực hiện rất

hiệu quả trên tập hợp các tập hợp mẩu tin có khóa. Chúng ta muốn xây dựng và bảo quản một cấu trúc dữ liệu chứa các mẩu tin có các khóa số và hỗ trợ vài thao tác sau:

- Tạo (construct) một hàng đợi có độ ưu tiên gồm N phần tử cho trước.
- Chèn (insert) một phần tử mới vào hàng đợi có độ ưu tiên.
- Hủy bỏ (remove) phần tử lớn nhất trong hàng đợi có độ ưu tiên.
- Thay thế (replace) phần tử lớn nhất bằng một phần tử mới (trừ khi phần tử mới lớn hơn)
- Thay đổi (change) độ ưu tiên của một phần tử
- Xóa (delete) một phần tử bất kỳ.
- Nối kết (join) hai hàng đợi có độ ưu tiên thành một hàng đợi lớn.

(Nếu các mẩu tin có các khóa trùng, chúng ta sẽ lấy "phần tử lớn nhất" nghĩa là mẩu tin nào có giá trị khóa lớn nhất)

Thao tác thay thế (replace) hầu như tương đương với chèn (insert) rồi mới hủy (remove) (khác nhau là chèn/hủy đòi hỏi hàng đợi ưu tiên phát triển tạm thời bằng một phần tử): chú ý là nó khác với thao tác hủy: rồi mới chèn. Thao tác này giống như một thao tác tách rời, vì một vài cài đặt của hàng đợi có độ ưu tiên có thể thực hiện thao tác thay thế rất hiệu quả. Tương tự, thao tác thay đổi (change) có thể được cài đặt như là thao tác xóa (delete) rồi mới chèn và tạo (construct) có thể được cài đặt bằng cách sử dụng nhiều lần thao tác chèn, nhưng những thao tác này có thể cài đặt trực tiếp có hiệu quả hơn khi biết chọn cấu trúc dữ liệu. Thao tác nối (join) yêu cầu cấu trúc dữ liệu cải tiến để cài đặt có hiệu quả; chúng ta sẽ tập trung vào một cấu trúc dữ liêu "cổ điển" gọi là heap mà cho phép cài đặt hiệu quả năm thao tác đầu tiên.

Hàng đợi có độ ưu tiên như đã mô tả ở trên là một ví dụ rất hay về cấu trúc dữ liệu trừu tượng (abstract data structure) như đã trình bày ở chương 3: nó định nghĩa rất tốt các thao tác thực hiện

độc lập với cách dữ liệu được tổ chức và xử lý trong bất kỳ cài đặt cụ thể nào.

Các phương pháp cài đặt khác nhau của hàng đợi có độ ưu tiên liên quan đến đặc điểm hoạt động của các thao tác khác nhau, đầu tiên là tổng chi phí. Thực vậy, khác biệt về thực hiện chính là khác biệt duy nhất theo quan điểm về cấu trúc dữ liệu trừu tượng. Trước hết, chúng ta giải thích quan điểm này bằng cách tìm hiểu một vài cấu trúc dữ liệu cơ bản để cài đặt hàng đợi có độ ưu tiên. Kế tiếp, chúng ta sẽ xem xét một cấu trúc dữ liệu nâng cao hơn và cách thức các thao tác khác nhau có thể được cài đặt hiệu quả khi dùng cấu trúc dữ liệu này. Chúng ta hãy để ý một thuật toán sắp xếp quan trọng theo sau các cài đặt này.

CÁC CÀI ĐẶT CƠ BẢN

Một cách để tổ chức một hàng đợi có độ ưu tiên mà tương tự như một xâu không có thứ tự là lưu giữ các phần tử trong mảng a[1..N] mà không cần chú ý đến khóa. Vì vậy tạo không phải là thao tác (no-op) cho cấu trúc này, nhưng chèn và hủy có thể cài đặt như sau:

```
procedure insert(v:integer);
   begin N:=N+1; a[N]:=v;
   end;
function remove:integer;
   var j,max:integer;
   begin
      max:=1;
      for j:=2 to N do if a[j]>a[max] then max:=j;
      remove:=a[max]; a[max]:=a[N];
      N:=N-1;
   end;
```

Để chèn chỉ cần tăng N và đặt phần tử mới vào a[N], đây là một thao tác cố định. Nhưng thao tác hủy cần duyệt qua mảng để tìm phần tử có khóa lớn nhất, nó cần thời gian tuyến tính (tất cả các phần tử trong mảng đều được kiểm tra), rồi hoán vị a[N] với phần tử có khóa lớn nhất và giảm N. Cài đặt của thao tác thay thế (replace) tương tự nên ta bỏ qua.

Để cài đặt thao tác thay đổi (thay đổi độ ưu tiên của phần tử trong a[k]), đơn giản có thể lưu trữ giá trị mới, còn để xóa phần tử trong a[k], có thể hoán vị nó với a[N] và giảm N xuống như trong dòng cuối của thao tác hủy. (Nhắc lại từ chương 3 là những thao tác như vậy mà liên quan đến các phần tử dữ liệu riêng lẽ, chỉ có ý nghĩa trong cài đặt "con trỏ" hay "gián tiếp" mà trong đó tham chiếu được bảo quản cho mỗi phần tử vào vị trí hiện hành trong cấu trúc dữ liệu).

Một cấu trúc cơ bản khác là sử dụng một xâu có thứ tự (ordered list), dùng lại một mảng a[1..N] nhưng giữ các phần tử theo thứ tự khóa tăng dần. Thao tác hủy liên quan đến việc trả về a[N] và giảm N, nhưng thao tác chèn liên quan đến việc chuyển các phần tử lớn hơn trong mảng sang bên phải một vị trí, thao tác này cần thời gian tuyến tính.

Bất kỳ thuật toán hàng đợi có độ ưu tiên nào cũng chuyển thành thuật toán sắp xếp bằng cách dùng thao tác chèn nhiều lần để xây dựng một hàng đợi có độ ưu tiên chứa tất cả các phần tử được sắp, rồi dùng thao tác hủy liên tục để làm rỗng hàng đợi có độ ưu tiên đưa vào các phần tử theo thứ tự ngược lại. Sử dụng một hàng đợi có độ ưu tiên được biểu diễn như một xâu chưa sắp thứ tự tương ứng với việc sắp bằng phương pháp chọn; sử dụng xâu có thứ tự tương ứng với việc sắp bằng phương pháp chèn.

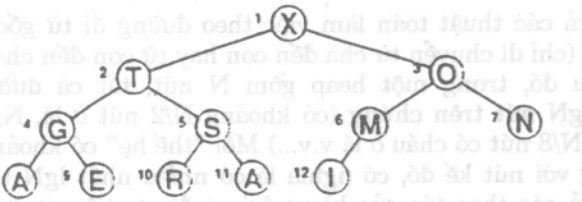
Các xâu liên kết cũng có thể được sử dụng cho xâu chưa sắp thứ tự hay xâu có thứ tự thay vì cài đặt mảng ở trên. Cấu trúc này không thay đổi đặc trưng cơ bản của các thao tác chèn, hủy, thay thế, nhưng có thể thay đổi trong thao tác xóa và nối kết.

Các cài đặt này không đưa ra ở đây vì nó tương tự với các thao tác xâu cơ bản trình bày ở chương 3, và vì các phép cài đặt của các phương pháp tương tự đối với bài toán tìm kiếm ở chương 14.

Thông thường, dễ nhớ các phép cài đặt đơn giản này bởi vì chúng thường phổ dụng hơn các phương pháp phức tạp trong nhiều trường hợp thực tế. Chẳng hạn, cài đặt của xâu chưa sắp thứ tự thích hợp trong một ứng dụng mà chỉ cần thực hiện vài thao tác "hủy phần tử lớn nhất" đối nghịch với một số thao tác chèn trong khi một xâu có thứ tự sẽ thích hợp nếu các phần tử cần chèn luôn có khuynh hướng gần với phần tử lớn nhất trong hàng đợi có độ ưu tiên.

CẤU TRÚC DỮ LIỆU HEAP

Cấu trúc dữ liệu mà chúng ta sử dụng để hỗ trợ cho thao tác hàng đợi có độ ưu tiên liên quan đến việc lưu trữ mẩu tin trong một mảng, bằng cách này mỗi khóa bảo đảm lớn hơn các khóa ở hai vị trí xác định khác. Lần lượt, mỗi khóa này phải lớn hơn hai khóa nữa. Thứ tự này rất dễ thấy nếu chúng ta dùng một mảng có cấu trúc cây hai chiều, như hình 11.1 mỗi khóa có hai đường thẳng xuống hai khóa khỏ hơn.



Hình 11.1 *Biểu diễn cây đầy đủ của một heap*

Nhắc lại từ chương 4 là cấu trúc này được gọi là "cây nhị phân đầy đủ"; nó có thể tạo ra bằng cách đặt một nút (gọi là gốc) và tiếp tục đi xuống từ trái sang phải, liên kết hai nút phía dưới, mỗi nút ở mức trên, cho đến khi đặt xong N nút, hai nút dưới mỗi nút được gọi là nút con của nó, nút ở mức trên mỗi nút gọi là nút cha của nó. Chúng ta cần các khóa trong cây thỏa điều kiện heap: khóa trong mỗi nút nên lớn hơn hay bằng các khóa của nút con của nó (nếu có). Chú ý là cây này sẽ thể hiện khóa lớn nhất ở gốc.

Chúng ta có thể biểu diễn cây nhị phân đầy đủ bên trong một mảng bằng cách đặt gốc ở vị trí 1, con của nó ở vị trí 2 và 3, các nút mức kế ở vị trí 4, 5, 6, 7, v.v... như các số trong hình 11.1. Ví dụ như biểu diễn mảng cho cây ở trên được trình bày ở hình 11.2

k	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*
a[k]	X	T	O	G	S	M	N	A	E	R	A	I

Hình 11.2 *Biểu diễn mảng của heap*

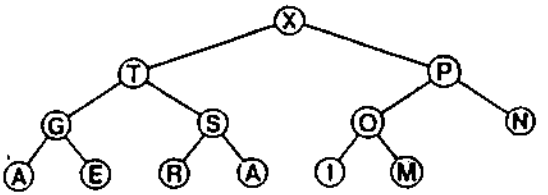
Biểu diễn tự nhiên này rất có ích vì rất dễ lấy dữ liệu ở vị trí j div 2 và ngược lại, hai nút con của một nút ở vị trí j sẽ ở vị trí 2j và 2j+1. Điều này làm cho việc di chuyển trên cấu trúc cây dễ dàng hơn nếu cây được cài đặt bằng biểu diễn liên kết chuẩn (với mỗi phần tử chứa một con trỏ đến cha và con của nó). Cấu trúc của cây nhị phân có đủ tính linh động để cho phép cài đặt các thuật toán hàng đợi có độ ưu tiên một cách hiệu quả. Một heap là một cây nhị phân đầy đủ được biểu diễn như một mảng, trong đó mỗi nút thỏa điều kiện heap. Đặc biệt khóa lớn nhất luôn ở vị trí đầu trong mảng.

Tất cả các thuật toán làm việc theo đường đi từ gốc đến đáy của heap (chỉ di chuyển từ cha đến con hay từ con đến cha). Rất dễ thấy điều đó, trong một heap gồm N nút, tất cả đường đi có khoảng lgN nút trên chúng (có khoảng N/2 nút ở lá, N/4 nút có con ở lá, N/8 nút có cháu ở lá v.v...) Mỗi "thế hệ" có khoảng nửa số nút bằng với nút kế đó, có nghĩa là có nhiều nhất lgN thế hệ. Vì vậy tất cả các thao tác của hàng đợi có độ ưu tiên (trừ join) cần thời gian logarit khi sử dụng cấu trúc heap.

CÁC THUẬT TOÁN TRÊN HEAP

Các thuật toán về hàng đợi có độ ưu tiên đều thực hiện bằng cách trước tiên có sự hiệu chỉnh về cấu trúc mà có thể vi phạm điều kiện heap, rồi di chuyển trên heap để hiệu chỉnh dần để thỏa điều kiện heap mọi nơi. Vài thuật toán đi lại trên heap từ cuối heap lên đến gốc, các thuật toán khác di chuyển từ gốc xuống cuối heap. Trong tất cả các thuật toán, giả thiết rằng các mẩu tin là các khóa số nguyên kích thước một từ lưu trong một mảng a có kích thước tối đa, còn kích thước hiện hành của heap lưu trong số nguyên N. Chú ý là N là một bộ phận định nghĩa của heap giống như khóa và mẩu tin vậy.

Để có thể tạo một heap, trước tiên cần cài đặt thao tác chèn. Vì thao tác này sẽ tăng kích thước của heap lên 1, nên N cũng phải tăng. Vì thế vi phạm tính chất của heap. Nếu tính chất heap bị phá vỡ (nút mới lớn hơn nút cha của nó, thì có thể hiệu chỉnh lại bằng cách hoán vị nút mới với nút cha của nó. Điều này lần lượt nảy sinh vi phạm tính chất của heap và vì vậy tiếp tục hiệu chỉnh



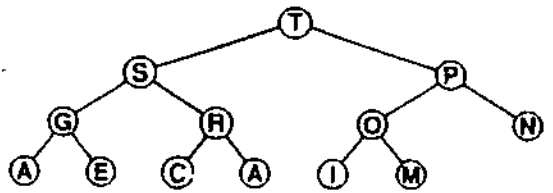
Hình 11.3 *Chèn một phần tử mới (P) vào heap*

tương tự như trên. Ví dụ, nếu P được chèn vào heap ở trên, trước hết lưu nó trong a[N] như là con phải của M. Kế đó, hoán vị nó với M vì nó lớn hơn M, và hoán vị với O vì P lớn hơn O, và quá trình vẫn kết thúc vì nó hơn X. Cấu trúc heap có kết quả trong hình 11.3.

Đoạn mã chương trình cho phương pháp này rất đơn giản. Trong cài đặt sau đây, thao tác chèn dùng để thêm một phần tử mới vào a[N], rồi gọi thủ tục upheap (N) để hiệu chỉnh sự vi phạm tính chất heap tại vị trí N:

```
procedure upheap(k:integer);
   var v:integer;
   begin
   v:=a[k]; a[0]:=maxint;
   while a[k div 2]<=v do
      begin a[k]:=a[k div 2]; k:=k div 2 end;
   a[k]:= v;
   end;
procedure insert(v:integer);
   begin
   N:=N+1; a[N]:=v; upheap(N);
   end;
```

Nếu k div 2 thay bằng k-1 ở bất kỳ đâu trong chương trình, chúng ta sẽ có một bước sắp bằng phương pháp chèn (cài đặt một hàng đợi có độ ưu tiên bằng xâu sắp thứ tự), nhưng ở đây chúng ta sẽ "chèn" phần tử mới dọc theo đường đi từ N lên đến gốc. Giống như phương pháp sắp chèn, không cần phải hoán vị tất cả trong vòng lặp, bởi vì v là biến trong phép hoán vị, một khóa cầm canh phải đặt ở a[0] để ngưng vòng lặp trong trường hợp v lớn hơn các khóa trên heap.



Hình 11.4 *Thay thế phần tử lớn nhất trong heap (bằng C)*

Thao tác thay thế liên quan đến việc thay thế ở gốc bằng một khóa mới rồi di chuyển từ gốc xuống cuối heap để phục hồi tính chất heap. Chẳng hạn như nếu X trong heap trên thay bằng C, bước đầu tiên là lưu trữ C ở gốc. Điều này sẽ vi phạm tính chất heap, nhưng có thể điều chỉnh bằng cách hoán vị C với T, là nút lớn hơn trong hai nút con của gốc. Lại tiếp tục vi phạm tính chất heap nữa ở mức kế, nên phải hoán vị C với nút lớn hơn trong hai nút con của nó (trong trường hợp này là S). Quá trình tiếp tục cho đến khi tính chất của heap không còn vi phạm nữa. Trong ví dụ, C tạo cho nó một đường đi đến cuối heap và đưa ra một heap được trình bày ở hình 11.4.

Thao tác "hủy phần tử lớn nhất" liên quan đến quá trình giống như vậy. Bởi vì heap sẽ cắt bớt một phần tử sau khi thao tác, nên cần giảm N, không có chỗ nào dành cho phần tử lưu ở vị trí cuối. Nhưng phần tử lớn nhất (nằm trong a[1] sẽ bị hủy, vì thế thao tác hủy sẽ chuyển thành thao tác thay thế cho nó phần tử nằm trong a[N]). Heap được trình bày ở hình 11.5 là kết quả của việc hủy T khỏi cấu trúc heap ở hình 11.4, bằng cách thay thế nó với M, rồi đi dần xuống để đưa phần tử lớn hơn trong hai con của nó lên, cho đến khi đến một nút có hai con nhỏ hơn M.

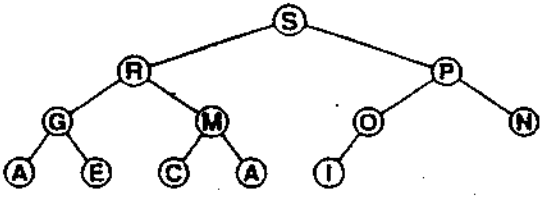
Cài đặt của hai thao tác này xoay quanh quá trình hiệu chỉnh heap mà luôn thỏa tính chất heap mọi nơi ngoại trừ ở gốc. Nếu khóa ở gốc quá bé, phải chuyển nó xuống mà không vi phạm tính chất của heap ở bất kỳ nút nào mà nó đến. Có nghĩa là cùng một thao tác có thể sử dụng để hiệu chỉnh heap sau khi giá trị nhỏ hơn ở bất kỳ vị trí nào. Có thể cài đặt như sau:

```
procedure downheap(k:integer);
   label 0;
   var i,j,v:integer;
   begin
   v:=a[k];
   while k<=N div 2 do
      begin
      j:=k+k;
      if j  then if a[j]<a[j+1] then j:=j+1;
      if v=a[j] then goto 0; a[k]:=a[j]; k:=j;
      end;
      0:a[k]:=v
   end;
```

Thủ tục này lần xuống trong heap, hoán vị nút ở vị trí k với nút lớn hơn trong hai nút con nếu cần và ngưng khi nút ở vị trí k lớn hơn hai nút con hay khi đến cuối heap (chú ý là đối với nút ở vị trí k có thể chỉ có một con: trường hợp này phải xử lý thích hợp). Như trên, không cần một hoán vị đầy đủ vì v luôn liên quan đến việc hoán vị. Vòng lặp bên trong của chương trình này là một ví dụ của vòng lặp thực sự có hai điểm thoát khác nhau: một cho trường hợp đến cuối heap (như trong ví dụ trên), và một cho trường hợp thỏa tính chất heap ở vị trí nào đó trên heap. Có thể tránh lệnh goto, trong vài trường hợp thì sẽ rõ ràng hơn.

Cài đặt của thao tác hủy là một ứng dụng trực tiếp của thủ tục trên:

```
function remove:integer;
   begin
      remove:=a[1]; a[1]:=a[N]; N:=N-1; downheap(1);
   end;
```



Hình 11.5 *Hủy phần tử lớn nhất trong heap*

Giá trị trả về được khởi động từ a[1] và sau đó phần tử trong a[N] được gán cho a[1] và kích thước của heap giảm xuống, chỉ có một lệnh gọi downheap để hiệu chỉnh tính chất của heap.

Cài đặt của thao tác thay thế phức tạp hơn một chút:

```
function replace(v:integer):integer;
  begin
  a[0]:=v; downheap(0); replace:=a[0];
  end;
```

Đoạn mã này dùng a[0] như sau: con của nó là 0 (chính nó) và 1, vì thế nếu v lớn hơn phần tử lớn nhất trong heap, không cần đụng đến heap ; mặt khác đặt v vào trong heap, và trả về a[1].

Thao tác xóa một phần tử bất kỳ ra khỏi heap và thao tác thay đổi cũng cần cài đặt bằng cách liên kết các phương pháp ở trên. Chẳng hạn, nếu tăng độ ưu tiên của phần tử tại ví trí k, thì gọi upheap(k) và ngược lại gọi downheap(k)

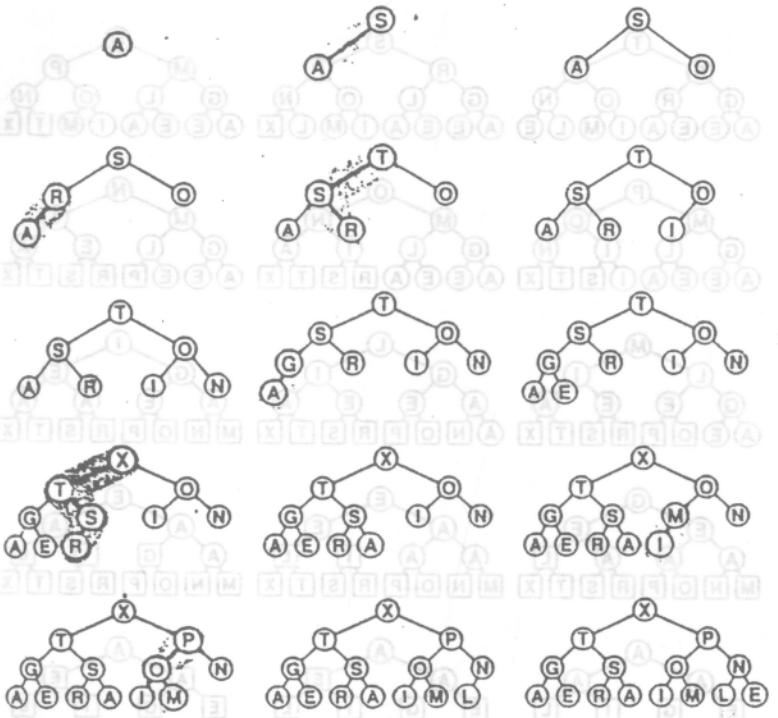
TÍNH CHẤT 11.1. *Tất cả các thao tác cơ bản như: chèn, hủy, thay thế, (downheap và upheap), xóa và thay đổi cần ít hơn 2lgN lần so sánh trên một heap có N phần tử.*

Tất cả các thao tác này liên quan đến việc di chuyển dọc theo đường từ gốc đến cuối heap cần không nhiều hơn lgN phần tử của một heap có kích thước N. Downheap cần hai lần so sánh trong vòng lặp của nó, thao tác khác chỉ cần lgN lần so sánh.

Chú ý rằng thao tác join không bàn đến trong xâu nầy. Thao tác này dường như cần cấu trúc dữ liệu phức tạp hơn. Mặt khác, trong nhiều ứng dụng, thao tác nầy được dùng ít hơn các thao tác khác.

HEAPSORT (phương pháp sắp xếp dùng heap)

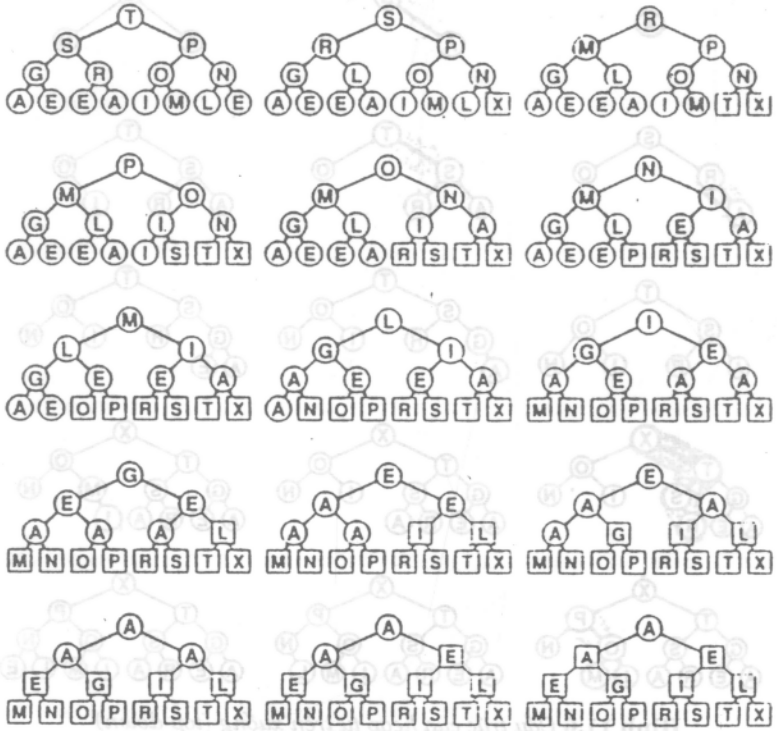
Một phương pháp sắp xếp hiệu quả có thể xác định dựa vào các thao tác heap cơ bản ở trên. Phương pháp này gọi là sắp xếp dùng heap (heapsort) nó không chiếm dụng thêm bộ nhớ và bảo đảm sắp M phần tử cần khoảng MlogM bước bất chấp dữ liệu nhập. Bất tiện là vòng lặp bên trong dài hơn vòng lặp bên trong của Quicksort và trung bình chậm hơn Quicksort hai lần.



Hình 11.6 *Cấu trúc của heap từ trên xuống (top-down)*

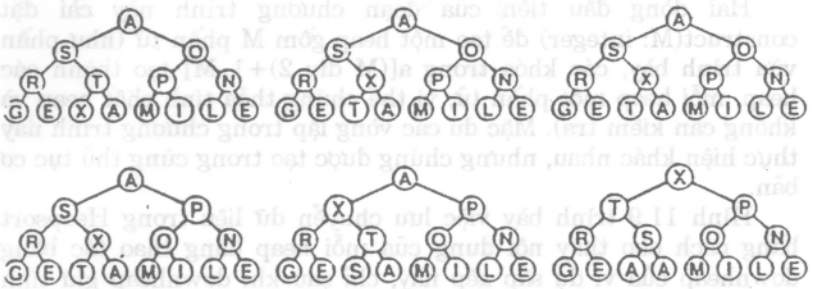
Tư tưởng là tạo một heap chứa các phần tử được sắp rồi hủy chúng theo thứ tự. Trong phần này, N vẫn là kích thước của heap vì thế chúng ta sẽ sử dụng M cho số phần tử đã được sắp. Một cách để sắp là cài đặt thao tác tạo bằng cách thực hiện M thao tác chèn, như trong hai dòng đầu của đoạn chương trình sau, rồi thực hiện M thao tác hủy:

```
N:=0;
for k:=1 to M do insert(a[k]);
for k:=M downto 1 do a[k]:=remove;
```



Hình 11.7 *Sắp xếp từ một heap*

Đoạn mã này sẽ phá vỡ các qui luật của cấu trúc dữ liệu trừu tượng bằng cách giả thiết một biểu diễn đặc biệt cho hàng đợi có độ ưu tiên (trong suốt vòng lặp, hàng đợi có độ ưu tiên tồn tại trong [1..k-1]) nhưng rất hợp lý để thực hiện điều này vì chúng ta đang cài đặt một phương pháp sắp xếp, không cài một hàng đợi có độ ưu tiên. Các thủ tục hàng đợi có độ ưu tiên chỉ dùng cho các mục đích mô tả; trong các phép cài đặt phương pháp sắp thực sự, chúng ta sử dụng đoạn mã chương trình từ thủ tục để tránh các lệnh gọi thủ tục không cần thiết.



Hình 11.8 *Cấu trúc của heap từ dưới lên (bottom-up)*

Hình 11.6 cho thấy heap được tạo khi chèn các khóa A S O R T I N G E X A M P L E theo thứ tự này vào một heap được khởi tạo rỗng, và hình 11.7 cho thấy các khóa này được sắp bằng cách hủy X, rồi hủy T, v.v...

Tạo một heap bằng cách đi ngược trở lại thì tốt hơn, tạo các heap nhỏ từ cuối heap lên, như đã trình bày ở hình 11.8. Phương pháp này trình bày các vị trí trong mảng như là gốc của các heap nhỏ và lợi dụng tính chất này là downheap sẽ thực hiện trên các heap nhỏ như heap lớn. Thực hiện hướng ngược lại trên heap mọi nút là gốc của heap có sắp thứ tự ngoại trừ gốc; downheap hoàn tất công việc thực hiện (không cần làm gì cả đối với heap có một phần tử, vì thế việc duyệt trên mảng chỉ thực hiện nửa đường).

Chúng ta lưu ý rằng thao tác hủy có thể cài đặt bằng cách hoán vị phần tử đầu và cuối, giảm N, và gọi downheap(1). Điều này đưa đến phép cài đặt sau của Heapsort:

```
procedure heapsort;
  var k,t:integer;
  begin
  N:=M;
  for k:=M div 2 downto 1 do downheap(k);
  repeat
    t:=a[1]; a[1]:=a[N]; a[N]:=t;
    N:=N-1; downheap(1);
  until N<=1;
  end;
```

Hai dòng đầu tiên của đoạn chương trình này cài đặt construct(M: integer) để tạo một heap gồm M phần tử (như phần vừa trình bày, các khóa trong a[(M div 2)+1..M] tạo thành các heap, mỗi heap một phần tử, vì thế chúng thỏa tính chất heap và không cần kiểm tra). Mặc dù các vòng lặp trong chương trình này thực hiện khác nhau, nhưng chúng được tạo trong cùng thủ tục cơ bản.

Hình 11.9 trình bày việc lưu chuyển dữ liệu trong Heapsort bằng cách cho thấy nội dung của mỗi heap đang thao tác bằng downheap của ví dụ sắp xếp này, chỉ sau khi downheap giữ tính chất heap thỏa mọi nơi.

TÍNH CHẤT 11.2. *Tạo heap từ dưới lên cần thời gian tuyến tính*

Tính chất này được khẳng định là do đa số các heap xử lý đều nhỏ. Chẳng hạn để tạo một heap gồm 127 phần tử, phương pháp này gọi downheap trên 64 heap có kích thước 1, 32 heap có kích thước 3, 16 heap có kích thước 7, 8 heap có kích thước 15, 4 heap có kích thước 31, 2 heap có kích thước 63, và 1 heap có kích thước 127,vì vậy tổng cộng cần

$$64.0 + 32.1 + 16.2 + 8.3 + 4.4 + 2.5 + 1.6 = 120$$

lần "đẩy" (bằng hai lần so sánh) trong trường hợp xấu nhất. Với $M = 2^m$, một chặn trên của số lần so sánh là:

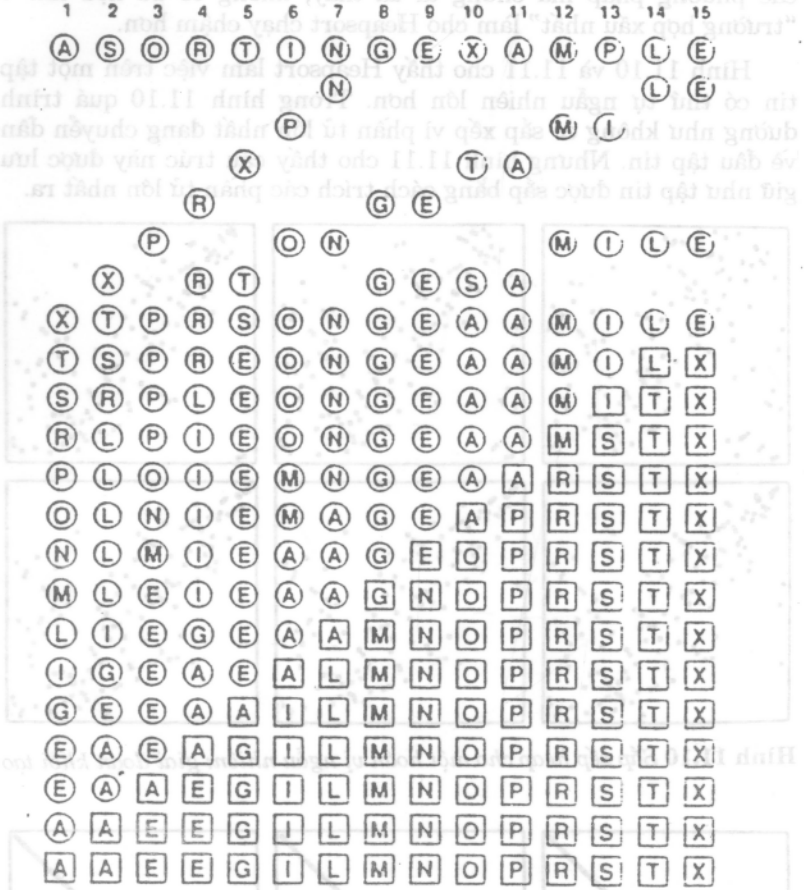
$$\sum_{1 \leqslant k \leqslant m} (k-1)2^{m-k} = 2^m - m - 1 < M$$

và chứng minh tương tự khi M không là lũy thừa của 2.

Tính chất này không quan trọng đối với Heapsort bởi vì thời gian vẫn còn trội hơn bởi MlogM lần dành cho sắp xếp, nhưng lại quan trọng đối với các ứng dụng dùng hàng đợi có độ ưu tiên khác mà thao tác tạo đưa ra một thuật toán dùng thời gian tuyến tính. Chú ý là tạo một heap có M thao tác chèn cần MlogM bước trong trường hợp xấu nhất (dù thời gian trung bình là tuyến tính).

TÍNH CHẤT 11.3 *Heapsort dùng ít hơn 2MlgM so sánh để sắp M phần tử*

Chặn trên 3MlgM được suy trực tiếp từ tính chất 11.1. Chặn



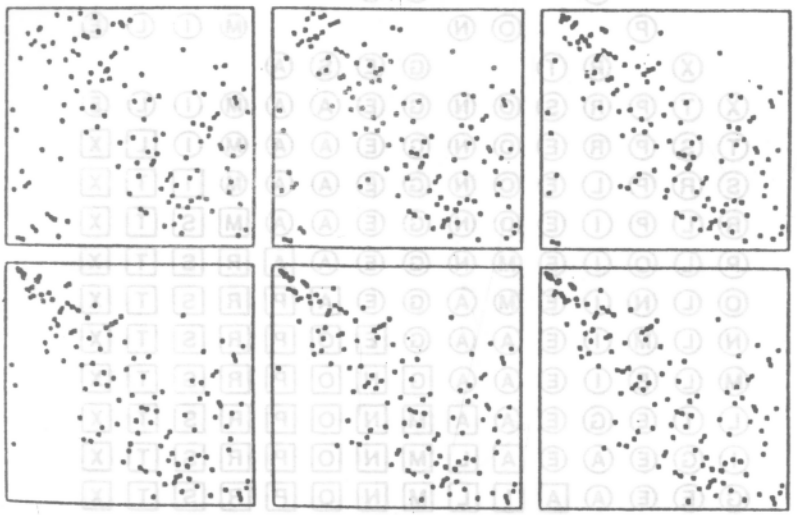
Hình 11.9 *Chuyển dữ liệu của Heapsort*

trên nêu ra trong tính chất này được dàn xếp cẩn thận hơn từ tính chất 11.2.

Như đã nói ở trên, tính chất 11.3 là lý do đơn giản mà Heapsort trở thành mối lưu tâm thực tế: số bước cần để sắp M phần tử tỷ lệ với MlogM, bất kể dữ liệu nhập. Điều này khác với

các phương pháp mà chúng ta đã thấy, không có dữ liệu nào ở "trường hợp xấu nhất" làm cho Heapsort chạy chậm hơn.

Hình 11.10 và 11.11 cho thấy Heapsort làm việc trên một tập tin có thứ tự ngẫu nhiên lớn hơn. Trong hình 11.10 quá trình dường như không có sắp xếp vì phần tử lớn nhất đang chuyển dần về đầu tập tin. Nhưng hình 11.11 cho thấy cấu trúc này được lưu giữ như tập tin được sắp bằng cách trích các phần tử lớn nhất ra.



Hình 11.10 *Sắp xếp heap cho một hoán vị ngẫu nhiên: giai đoạn khởi tạo*

Hình 11.11 *Sắp xếp heap cho một hoán vị ngẫu nhiên: giai đoạn sắp xếp*

CÁC HEAP GIÁN TIẾP

Trong nhiều ứng dụng của hàng đợi có độ ưu tiên, chúng ta không cần các mẩu tin phải di chuyển đi đâu cả. Thay vì vậy, chúng ta cần chương trình hàng đợi có độ ưu tiên không trả về giá trị ngoại trừ cho biết mẩu tin nào là mẩu tin lớn nhất v.v..., gần với quan điểm "sắp gián tiếp" hay "sắp dùng con trỏ" được trình bày ở chương 8. Sửa đổi các chương trình trên để thực hiện theo cách này là rất đúng cho dù đôi khi gây nên lẫn lộn. Việc kiểm chứng điều này chi tiết hơn ở đây là nên làm do rất tiện sử dụng heap.

Như ở chương 8, thay vì sắp xếp lại các khóa trong mảng a, chương trình hàng đợi có độ ưu tiên sẽ thực hiện với một mảng p phần tử bên trong mảng a sao cho a[p[k]] là mẩu tin tương ứng với phần tử thứ k của heap với k nằm giữa 1 và N (chúng ta vẫn tiếp tục giả thiết các mẩu tin có khóa một từ như ở chương 8, một tiện lợi cơ bản để mà có thể chuyển sang các mẩu tin và khóa phức tạp hơn). Hơn nữa, chúng ta cần dùng một mảng q để giữ vị trí heap của phần tử thứ k của mảng. Đây là cơ chế cho phép thao tác thay đổi và xóa. Vì vậy đầu vào của q cho phần tử lớn nhất trong mảng là 1 và cứ như vậy tiếp tục. Chẳng hạn, nếu chúng ta muốn thay đổi giá trị của a[k] chúng ta có thể tìm vị trí heap của nó trong q[k] và dùng upheap và downheap. Hình 11.12 cho thấy các giá trị trong mảng này trong ví dụ đang xét, chú ý là p[q[k]]=q[p[k]]=k đối với k có giá trị từ 1 đến N.

k	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
a[k]	A	S	O	R	T	I	N	G	E	X	A	M	P	L	E
p[k]	10	5	13	4	2	3	7	8	9	1	11	12	6	14	15
a[p[k]]	X	T	P	R	S	O	N	G	E	A	A	M	I	L	E
q[k]	10	5	6	4	2	13	7	8	9	1	11	12	3	14	15

Hình 11.12 *Các cấu trúc dữ liệu heap gián tiếp*

Chúng ta bắt đầu với p[k]=q[k]=k từ 1 đến N mà chứng tỏ là không phải thực hiện sắp xếp gì cả. Đoạn mã chương trình cho cấu trúc heap rất giống như trước đây:

```
procedure pqconstruct;
   var k:integer;
   begin
   N:=M;
   for k:=1 to N do
      begin
      p[k]:=k; q[k]:=k
      end;
   for k:=M div 2 downto 1 do pqdownheap(k);
   end;
```

Chúng ta sẽ định trước cài đặt chương trình hàng đợi có độ ưu tiên dựa vào heap gián tiếp có "pq" chỉ danh khi dùng chúng ở chương sau.

Bây giờ để sửa đổi thủ tục downheap để làm việc một cách gián tiếp, chúng ta chỉ cần kiểm tra nơi mà nó tham chiếu a. Ở nơi mà nó đã thực hiện một phép so sánh trước đó, nó phải truy xuất gián tiếp a nhờ vào p. Ở nơi mà nó đã thực hiện một phép di chuyển trước đó, nó phải di chuyển trong p, chứ không phải a, và nó phải sửa đổi q một cách tương ứng. Điều này dẫn đến cài đặt sau đây:

```
procedure pqdownheap(k:integer);
   label 0;
   var j, v:integer;
   begin
   v:=p[k];
   while k<=N div 2 do
      begin
      j:=k+k;
      if j<N then if a[p[j]] < a[p[j+1]] then j:=j+1;
      if a[v]=a[p[j]] then goto 0; p[k]:=p[j];
      q[p[j]]:=k; k:=j;
      end;
   0: p[k]:=v; q[v]:=k;
   end;
```

Các thủ tục khác nêu ở trên có thể sửa đổi theo cách tương tự để cài đặt "pqinsert", "pqchange", v.v...

Một phép cài đặt gián tiếp tương tự có thể phát triển dựa trên duy trì p như một mảng các con trỏ trên các mẩu tin được định vị riêng. Trong trường hợp này, mất công một chút để cài đặt hàm q (tìm vị trí heap cho mẩu tin cần tìm)

CÁC CÀI ĐẶT CẢI TIẾN

Nếu thao tác nối kết thực hiện hiệu quả, thì cài đặt mà chúng ta đã thực hiện không đủ và cần nhiều kỹ thuật cải tiến hơn. Mặc dù chúng không tập trung vào các chi tiết của phương pháp này, chúng ta vẫn có vài nhận xét trong phần thiết kế.

Thao tác nối kết nên thực hiện tốn khoảng thời gian bằng với các thao tác khác. Điều này hình thành biểu diễn không có dấu nối cho heap mà chúng ta đang sử dụng, vì hai heap lớn có thể bằng cách chuyển tất cả các phần tử ít nhất từ một trong hai heap vào một mảng lớn. Thực sự thì dễ dàng đưa thuật toán mà chúng ta đang kiểm tra để sử dụng biểu diễn sự kết nối. Đôi khi có các lý do khác để thực hiện (chẳng hạn, rất bất tiện để sử dụng một mảng liên tục có kích thước lớn). Trong biểu diễn nối kết trực tiếp phải giữ trong mỗi nút trỏ đến cả cha và lẫn con.

Điều kiện của heap hình như quá mạnh để cho phép cài đặt hiệu quả thao tác nối kết. Cấu trúc dữ liệu nâng cao được thiết kế để giải quyết vấn đề này là tất cả các cấu trúc yếu hơn hoặc là điều kiện cân bằng theo thứ tự để cần có tính linh động cho thao tác nối kết. Cấu trúc này cho phép tất cả các thao tác hoàn tất theo thời gian logarit.

BÀI TẬP

1. Vẽ heap có được khi các thao tác sau được thực hiện trên một heap rỗng ban đầu: insert(10), insert(5), insert(2), replace(4), insert(6), insert(8), remove, insert(7), insert(3)

2. Một tập tin sắp theo thứ tự ngược có phải là một heap không?

3. Hãy cho heap được tạo bởi việc áp dụng liên tiếp phép chèn trên các khóa E A S Y Q U E S T I O N.

4. Các vị trí nào có thể bị chiếm bởi khóa nhỏ thứ ba trong một heap kích thước 32.

5. Tại sao không dùng một biến cầm canh để tránh phép kiểm tra j<N trong downheap?

6. Hãy minh họa làm thế nào để nhận được các hàm của ngăn xếp và hàng đợi chuẩn như là trường hợp đặc biệt của các hàng đợi có ưu tiên.

7. Số khóa tối thiểu phải được di chuyển trong một thao tác "hủy cái lớn nhất" trong 1 heap là bao nhiêu? Hãy vẽ 1 heap có kích thước 15 mà số tối thiểu đạt được.

8. Viết một chương trình để xóa phần tử ở vị trí d trong một heap.

9. Hãy so sánh theo kinh nghiệm việc xây dựng heap từ dưới lên với việc xây dựng heap từ trên xuống, bằng cách tạo các heap với 1000 khóa ngẫu nhiên.

10. Hãy cho nội dung của mảng q sau khi pqconstruct được dùng trên các khóa E A S Y Q U E S T I O N.

12
SẮP XẾP BẰNG TRỘN

Trong chương 9 chúng ta đã tìm hiểu thao tác chọn, tìm phần tử nhỏ nhất thứ k trong tập tin. Chúng ta đã thấy việc chọn gần giống với việc chia một tập tin thành hai phần, k phần tử nhỏ nhất và N-k phần tử lớn nhất. Trong chương này chúng ta xem xét một quá trình bổ sung cho nó, trộn (merging), thao tác nối hai tập tin được sắp thành một tập tin được sắp lớn hơn. Như chúng ta đã biết, trộn là thao tác cơ bản của thuật toán sắp xếp đệ quy.

Chọn và trộn là các thao tác bổ sung lẫn nhau, có nghĩa là chọn chia một tập tin thành hai tập tin độc lập và trộn tổ hợp hai tập tin này thành một tập tin. Quan hệ giữa các thao tác này cũng dễ thấy nếu ta thử áp dụng hình thức "chia để trị" để tạo một phương pháp sắp xếp. Tập tin cũng có thể sắp xếp lại để cho khi sắp xong hai phần thì toàn bộ tập tin sẽ sắp xong hay phân thành hai phần đã được sắp rồi nối lại để tạo thành một tập tin được sắp trọn vẹn. Chúng ta đã xem điều gì xảy ra trong thể hiện đầu tiên: đó là Quicksort, cơ bản nó chứa một thủ tục chọn, theo sau là hai thủ tục đệ quy. Dưới đây chúng ta sẽ xem Mergesort, một bổ sung của Quicksort trong đó chứa hai lệnh gọi đệ quy, theo sau là một thủ tục trộn.

Sắp xếp bằng trộn, giống như sắp xếp bằng heap (Heapsort) có tiện lợi là nó sắp một tập tin gồm N phần tử cần thời gian tỉ lệ với NlogN ngay cả trường hợp xấu nhất. Bất lợi của sắp xếp bằng trộn là bộ nhớ cần dùng thêm tỉ lệ với N, trừ phi ta cố gắng khắc phục khó khăn này. Chiều dài của vòng lặp bên trong cỡ khoảng giữa của Quicksort và Heapsort, vì thế Mergesort là **ứng cử viên nặng** ký nếu lấy tốc độ làm mặt mạnh, đặc biệt nếu có sẵn chỗ trống dành riêng. Hơn nữa sắp xếp bằng trộn có thể cài đặt được sao cho

nó truy cập dữ liệu cơ bản theo phương pháp tuần tự (hết phần tử này đến phần tử khác) và điều này đôi khi là một thuận lợi rõ ràng. Chẳng hạn, sắp bằng trộn là phương pháp chọn lựa để sắp một xâu liên kết do truy xuất tuần tự là hình thức truy xuất duy nhất. Tương tự, xem trong chương 13, trộn là phương pháp cơ bản để sắp trên thiết bị truy xuất tuần tự, đâu cho các phương pháp sử dụng trong tình huống khác với các tình huống sử dụng cho sắp bằng trộn.

TRỘN

Trong nhiều môi trường xử lý dữ liệu, một tập tin dữ liệu lớn (đã được sắp) được duy trì cùng với các đầu vào mới được thêm một cách thường xuyên. Điển hình, một số dữ liệu nhập mới được nối vào cuối tập tin chính và toàn bộ tập tin phải được sắp lại. Trường hợp này dẫn đến ý tưởng là phải xây dựng một chiến lược để sắp một lô các dữ liệu nhập, rồi trộn nó vào tập tin chính. Trộn có nhiều ứng dụng tương tự khác rất đáng để tìm hiểu. Chúng ta cũng tìm hiểu một phương pháp sắp xếp dựa trên trộn.

Trong chương này chúng ta sẽ tập trung xét các chương trình để trộn-hai-đường (two-way merging): các chương trình nối hai tập tin nhập đã được sắp thành một tập tin xuất đã được sắp. Trong chương kế, chúng ta sẽ tìm hiểu chi tiết ở trộn-nhiều-đường (multiway merging) khi có nhiều hơn hai tập tin được sắp (ứng dụng quan trọng nhất của việc trộn bằng nhiều đường là sắp xếp ngoài, chủ đề chính của chương đó).

Giả sử chúng ta có hai mảng số nguyên đã sắp a[1..M] và b[1..N] mà chúng ta cần trộn thành một mảng thứ ba c[1..M+N]. Sau đây là cài đặt trực tiếp của chiến lược làm sao chọn cho c các phần tử nhỏ nhất từ a và b:

```
i:=1; j:=1;
a[M+1]:=maxint; b[N+1]:=maxint;
for k:=1 to M+N do
    if a[i] < b[j]
        then begin c[k]:=a[i]; i:=i+1 end
        else begin c[k]:=b[j]; j:=j+1 end;
```

Đơn giản hoá cài đặt bằng cách tạo chỗ trống lưu trữ trong mảng a và b cho các khoá cầm canh có các giá trị lớn hơn các khoá khác. Khi mảng a (hay b) cạn, vòng lặp chỉ đơn giản chuyển phần còn lại của b (hay a) vào mảng c. Phương pháp này dĩ nhiên dùng M+N lần so sánh. Nếu a[M+1] và b[N+1] không dùng được cho khoá cầm canh thì cần kiểm tra để chắc chắn là i luôn nhỏ hơn M và j luôn luôn nhỏ hơn N. Cách khác để tránh được khó khăn này sẽ được dùng dưới đây cho bản cài đặt của Mergeshort.

Thay vì dùng thêm chỗ trống tỉ lệ với kích thước của tập tin cần trộn, ta mong muốn có một phương pháp tại-chỗ (in-place), dùng c[1..M] cho một đầu vào và c[M+1..M+N] cho một cái khác. Đầu tiên rất khó khẳng định là điều này có thể thực hiện dễ dàng, và các phương pháp như vậy sẽ tồn tại, nhưng chúng thì quá phức tạp mà ngay cả một phép sắp xếp tại chỗ rất có thể sẽ là hiệu quả hơn trừ phi ta quan tâm đến chúng thật nhiều. Chúng ta sẽ quay lại vấn đề này dưới đây.

Vì cần thêm chỗ trống cho cài đặt thực tế, nên chúng ta chọn cài đặt bằng xâu liên kết. Thực sự, phương pháp này rất thích hợp với xâu liên kết. Một cài đặt đầy đủ minh họa tất cả các quy ước chúng ta sẽ sử dụng, chú ý là đoạn chương trình dành để trộn thực sự cũng đơn giản như đoạn mã ở trên:

```
type link = ↑ node;
   node = record key:integer; next:link end;
   var t,z:link;
   N:integer;
function merge (a,b:link):link;
   var c:link;
   begin
      c:=z;
      repeat
         if a↑.key <=b↑.key
         then begin c↑.next:=a; c:=a; a:=a↑.next end
         else begin c↑.next:=b; c:=b; b:=b↑.next end
      until c↑.key = maxint;
      merge:=z↑.next; z↑.next:=z;
   end;
```

Chương trình này trộn xâu do a trỏ đến với xâu do b trỏ đến, với sự trợ giúp của con trỏ c. Các danh sách được giả thiết là có nút "đuôi" (tail) như đã trình bày ở chương 3: tất cả các xâu kết thúc bằng nút z, thông thường nút này trỏ đến chính nó và cũng có vai trò như một lính canh, với z↑.k = maxint; Trong lúc trộn, sử dụng z để lưu đầu của xâu mới trộn (tương tự với cài đặt của readlist) và c trỏ đến cuối xâu mới trộn (nút có trường liên kết phải được thay đổi để thêm một phần tử mới vào xâu). Sau khi xây dựng xong xâu trộn, con trỏ đến nút đầu tiên của nó được lấy từ z và z được đặt lại để trỏ đến chính nó.

So sánh khóa trong khi trộn bao gồm cả trường hợp bằng nhau để cho việc trộn ổn định, nếu xâu b được xem như theo sau xâu a. Dưới đây, chúng ta sẽ thấy tính ổn định này trong quá trình trộn có liên quan đến tính ổn định trong các chương trình sắp xếp có sử dụng trộn là như thế nào.

SẮP XẾP TRỘN (Mergesort)

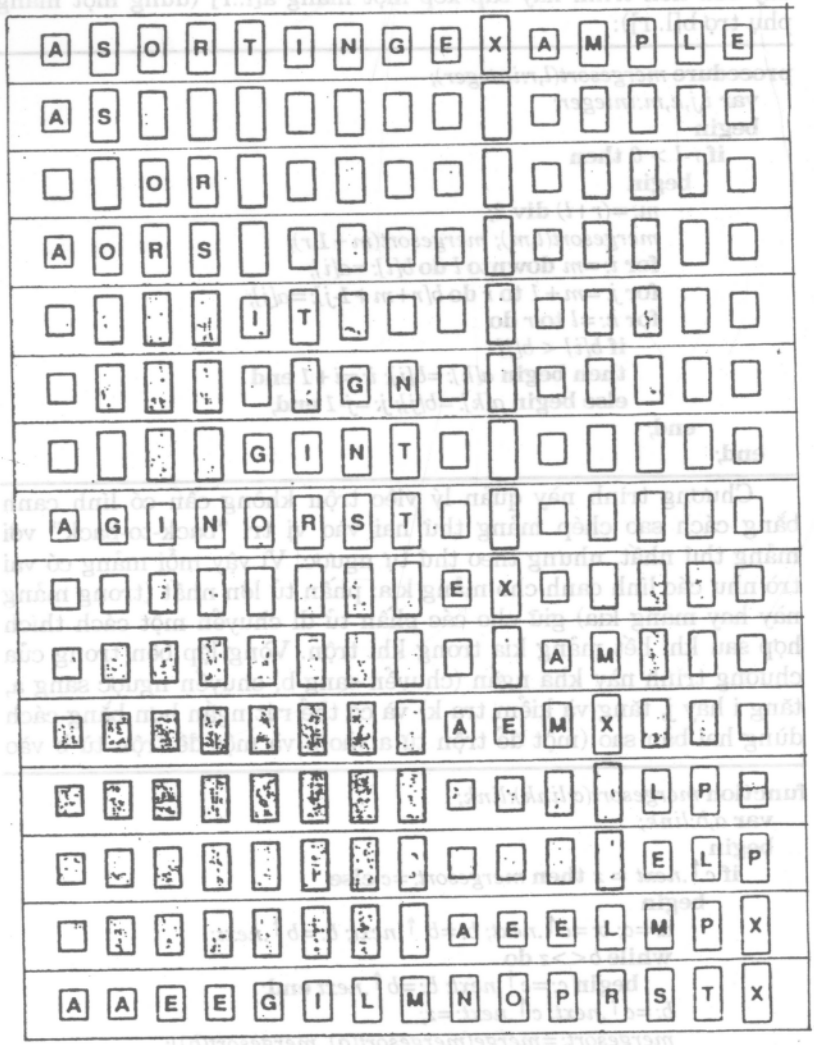
Một khi chúng ta có một thủ tục trộn, thì sẽ không khó khăn gì để dùng nó như một cái nền cho một thủ tục sắp đệ quy. Để sắp một tập tin cho trước, ta chia nó ra làm hai, sắp xếp hai nửa này (một cách đệ quy), sau đó trộn hai nửa này lại với nhau. Bản cài đặt sau

đây của tiến trình này sắp xếp một mảng a[l..r] (dùng một mảng phụ trợ b[l..r]):

```
procedure mergesort(l,r:integer);
   var i,j,k,m:integer;
   begin
      if r-l > 0 then
         begin
            m:=(r+l) div 2;
            mergesort(l,m); mergesort(m+1,r);
            for i:=m downto l do b[i]:=a[i];
            for j:=m+1 to r do b[r+m+1-j]:=a[j];
            for k:=l to r do
               if b[i] < b[j]
               then begin a[k]:=b[i]; i:=i+1 end
               else begin a[k]:=b[j]; j:=j-1 end;
         end;
   end;
```

Chương trình này quản lý việc trộn không cần có lính canh bằng cách sao chép mảng thứ hai vào vị trí "back-to-back" với mảng thứ nhất, nhưng theo thứ tự ngược. Vì vậy mỗi mảng có vai trò như các lính canh cho mảng kia: phần tử lớn nhất (trong mảng này hay mảng kia) giữ cho các phần tử di chuyển một cách thích hợp sau khi hết mảng kia trong khi trộn. Vòng lặp bên trong của chương trình này khá ngắn (chuyển sang b, chuyển ngược sang a, tăng i hay j, tăng và kiểm tra k) và có thể rút ngắn hơn bằng cách dùng hai bản sao (một để trộn từ a vào b và một để trộn từ b vào

```
function mergesort(c:link):link;
   var a,b:link;
   begin
      if c↑.next = z then mergesort:=c else
         begin
            a:=c; b:=c↑.next; b:=b↑.next; b:=b↑.next;
            while b<>z do
               begin c:=c↑.next; b:=b↑.next end;
            b:=c↑.next; c↑.next:=z;
            mergesort:=merge(mergesort(a), mergesort(b));
         end;
   end;
```



Hình 12.1 *Sắp xếp trộn đệ quy*

a), mặc dù điều này sẽ cần phải quay trở lại phần tử lính canh.

Tập tin chứa các khoá ví dụ được xử lý của chúng ta được minh họa ở hình 12.1 trang sau. Mỗi dòng cho thấy kết quả của lệnh gọi tới merge. Trước hết ta trộn A và S để có A S, rồi trộn O và R để có O R và dùng chúng với A S để có A O S R. Sau đó chúng ta trộn I T với G N để có G N I T, và trộn chúng với A O R S để có A G I N O R S T, ... Vì vậy phương pháp này tạo đệ quy từ các tập tin đã sắp có kích thước nhỏ thành các tập tin lớn hơn.

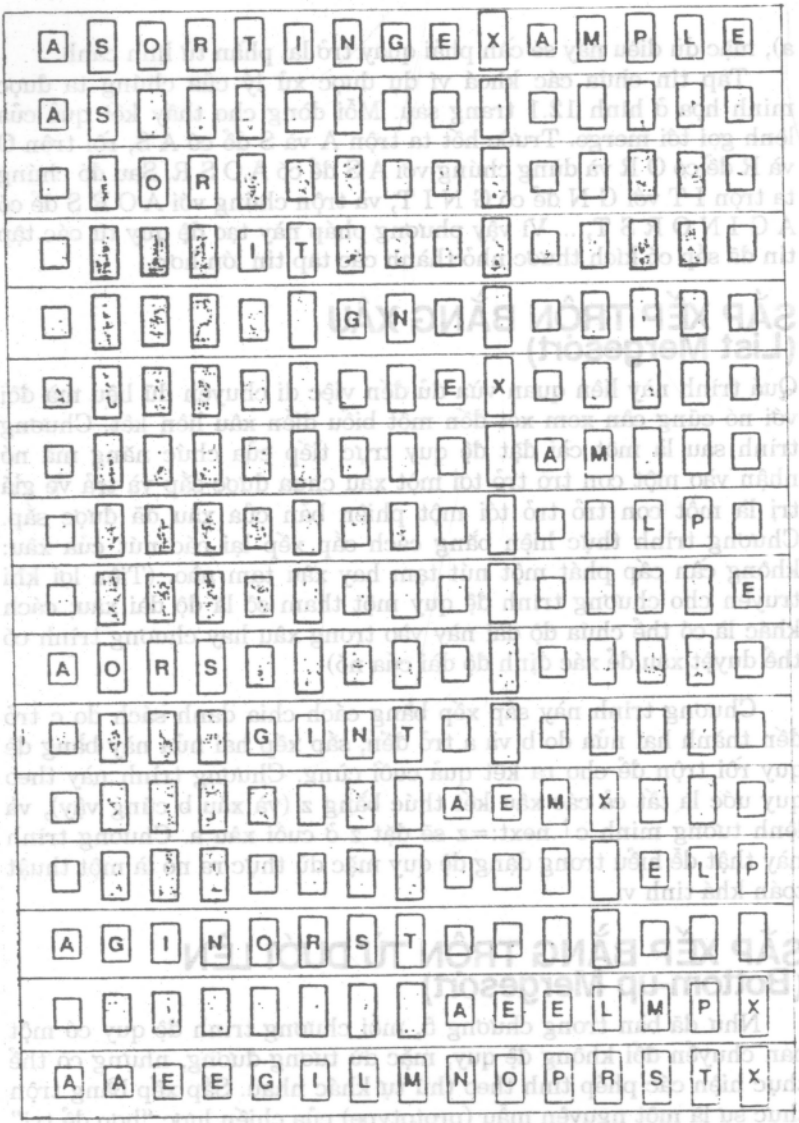
SẮP XẾP TRỘN BẰNG XÂU (List Mergesort)

Quá trình này liên quan vừa đủ đến việc di chuyển dữ liệu mà đối với nó cũng cần xem xét đến một biểu diễn xâu liên kết. Chương trình sau là một cài đặt đệ quy trực tiếp của chức năng mà nó nhận vào một con trỏ trỏ tới một xâu chưa được sắp và trả về giá trị là một con trỏ trỏ tới một phiên bản của xâu đã được sắp. Chương trình thực hiện bằng cách sắp xếp lại các nút của xâu: không cần cấp phát một nút tạm hay xâu tạm nào. (Tiện lợi khi truyền cho chương trình đệ quy một tham số là độ dài xâu; cách khác là có thể chứa độ dài này vào trong xâu hay chương trình có thể duyệt xâu để xác định độ dài của nó).

Chương trình này sắp xếp bằng cách chia danh sách do c trỏ đến thành hai nửa do b và a trỏ đến, sắp xếp hai nửa này bằng đệ quy rồi trộn để cho ra kết quả cuối cùng. Chương trình này theo quy ước là tất cả các xâu kết thúc bằng z (và xâu b cũng vậy), và lệnh tường minh c↑.next:=z sẽ đặt z ở cuối xâu a. Chương trình này thật dễ hiểu trong dạng đệ quy mặc dù thực ra nó là một thuật toán khá tinh vi.

SẮP XẾP BẰNG TRỘN TỪ DƯỚI LÊN (Bottom-up Mergesort)

Như đã bàn trong chương 5, mỗi chương trình đệ quy có một bản chuyển đổi không đệ quy, mặc dù tương đương, nhưng có thể thực hiện các phép tính theo thứ tự khác nhau. Sắp xếp bằng trộn thực sự là một nguyên mẫu (prototype) của chiến lược "hợp để trị" (combine and conquer) mà nó đặc trưng cho nhiều tính toán như



Hình 12.2 *Sắp xếp trộn không đệ quy*

vậy, và đáng nghiên cứu các bản cài đặt không đệ quy của nó một cách chi tiết.

Phiên bản không đệ quy đơn giản nhất của sắp bằng trộn xử lý trên tập hợp các tập tin khác nhau theo thứ tự: trước hết duyệt xâu, thực hiện việc trộn 1 với 1 để cho ra xâu con có kích thước 2 được sắp, sau đó duyệt xâu, thực hiện việc trộn 2 với 2 để cho ra xâu con có kích thước 4 được sắp, rồi lại thực hiện các phép trộn 4 với 4 để có xâu con kích thước 8 được sắp, ... cho đến khi toàn bộ xâu đã được sắp xong.

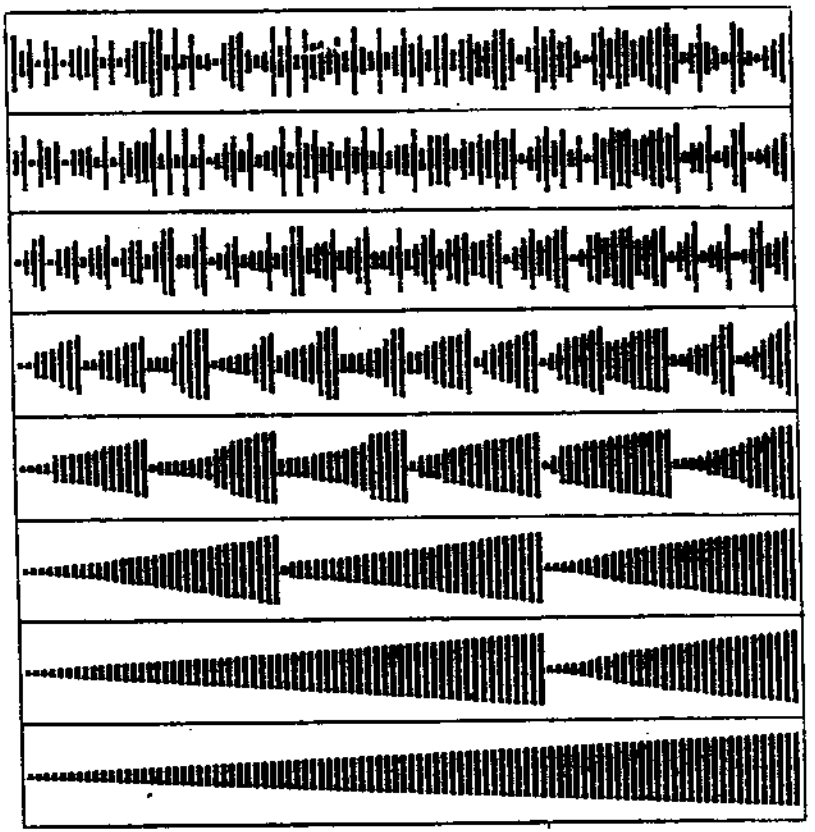
Quan trọng cần lưu ý là các phép trộn thực sự được tạo bởi phương pháp "dưới lên" này đều không giống như các phép trộn đã được thực hiện bởi các cài đặt đệ quy ở trên. Hãy xem quá trình sắp 95 phần tử ở hình 12.3. Lần trộn cuối cùng là 64 trộn 31, trong khi trong phép trộn đệ quy thì nó là 47 trộn 48. Tuy nhiên có thể sắp xếp sao cho các dãy tham gia trộn trong hai cách đều giống nhau, cho dù không có lý do cụ thể nào để làm như vậy cả.

Một cài đặt chi tiết của tiếp cận từ dưới lên, dùng xâu liên kết được cho dưới đây:

```
function mergesort (c:link):link;
  var a,b,head,todo,t:link;
    i,N:integer;
  begin
    N:=1; new(head); head↑.next:=c;
    repeat
      todo:=head↑.next; c:=head;
      repeat
        t:=todo; a:=t;
        for i:=1 to N-1 do t:=t↑.next;
        b:=t↑.next; t↑.next:=z; t:=b;
        for i:=1 to N-1 do t:=t↑.next;
        todo:=t↑.next; t↑.next:=z; c↑.next:=merge(a,b);
        for i:=1 to N+N do c:=c↑.next
      until todo = z;
      N:=N+N;
    until a=head↑.next;
    mergesort:=head↑.next
  end;
```

Chương trình này dùng một nút "đầu xâu" (do head trỏ đến),

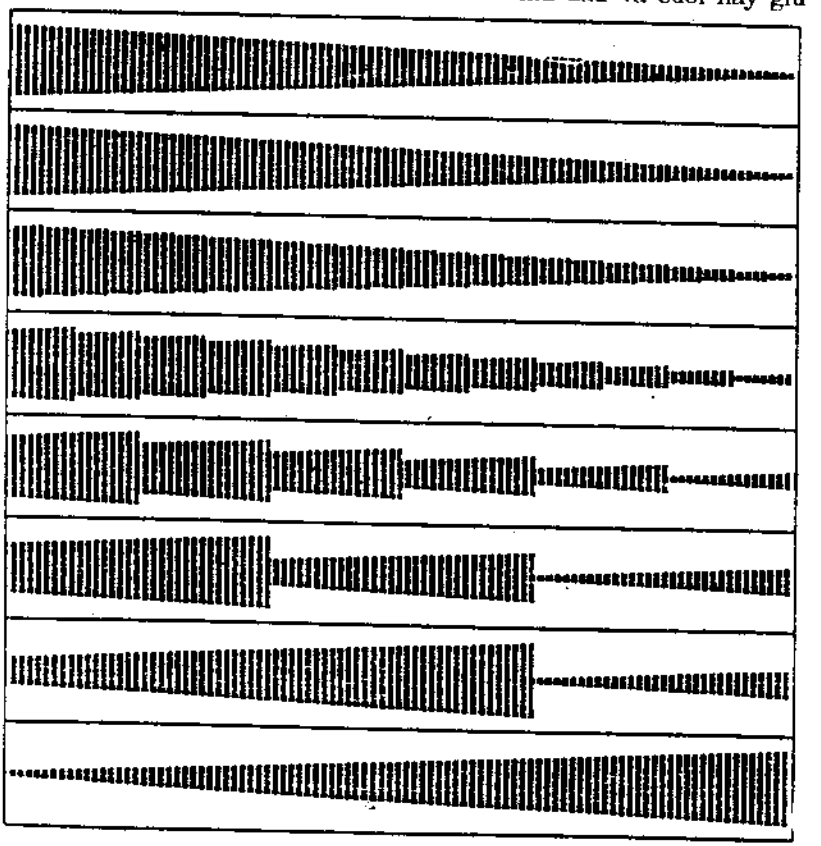
nút này có trường kết nối trỏ đến tập tin cần sắp. Mỗi lần lặp của vòng lặp repeat ngoài sẽ duyệt qua tập tin, cho ra một xâu liên kết gồm các tập tin con đã được sắp, dài gấp hai so với lần lặp trước. Tiến trình này được thực hiện bằng cách dùng hai con trỏ, một con trỏ trỏ đến phần của xâu chưa thấy (todo) và một con trỏ trỏ đến cuối phần đó của xâu của các tập tin con đã trộn xong (c). Vòng lặp repeat bên trong trộn hai tập tin con có chiều dài N bắt đầu ở nút do todo trỏ tới, nó cho ra một tập tin con có chiều dài



Hình 12.3 *Sắp xếp bằng trộn một chuyển vị ngẫu nhiên*

N+N được nối kết trên xâu kết quả c.

Cách trộn thực sự là lưu trữ một liên kết (link) với tập tin con đầu tiên được trộn trong a, rồi nhảy qua N nút (dùng mối liên kết tạm thời t), nối z vào cuối xâu của a, rồi thực hiện tương tự để có một xâu khác gồm N nút do b trỏ đến (cập nhật todo bằng một mối liên kết của nút cuối vừa thăm) và sau đó gọi merge (sau đó c được cập nhật bằng cách duyệt dần xuống cuối của xâu vừa trộn. Đây là phương pháp đơn giản hơn (nhưng ít hiệu quả hơn), chẳng hạn merge trả về các con trỏ trỏ đến cả hai đầu và cuối hay giữ



Hình 12.4 *Sắp xếp trộn một chuyển vị thứ-tự-ngược*

nhiều con trỏ trong mỗi nút của xâu).

Sắp xếp bằng trộn từ dưới lên cũng là phương pháp lý thú để dùng cho một cài đặt bằng mảng; phần này như một bài tập dành cho người đọc.

Tính chất 12.1 *Sắp xếp bằng trộn cần khoảng NlgN lần so sánh để sắp bất kỳ một tập tin nào gồm N phần tử.*

Trong cài đặt ở trên, mỗi phép trộn M trộn N cần M+N lần so sánh (điều này có thể biến thiên khoảng một hay hai tùy vào các biến cảm canh được sử dụng như thế nào). Đối với sắp xếp trộn từ dưới lên, cần lgN lần lặp, mỗi lần cần khoảng N lần so sánh. Đối với bản đệ quy, số lần so sánh được mô tả bằng quy nạp $M_N = 2M_{N/2} + N$ với $M_1 = 0$. Chúng ta biết ở chương 6 có giải pháp $M_N \approx N \lg N$. Các đối số này rất chính xác nếu N là lũy thừa của 2; phần này coi như bài tập để chứng minh là chúng đúng với N tổng quát. Hơn nữa, chúng cũng đúng trong trường hợp trung bình.

Tính chất 12.2 *Sắp xếp bằng trộn sử dụng một chỗ trống phụ trợ tỉ lệ với N.*

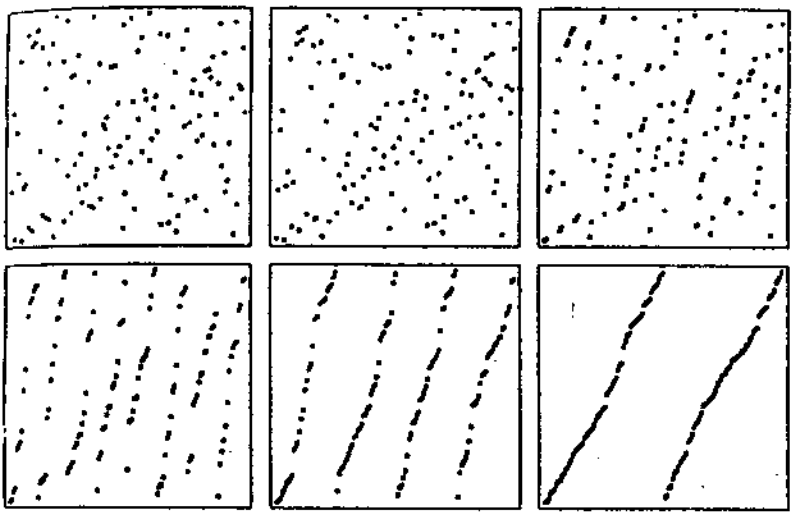
Điều này thấy rõ ràng khi cài đặt. Dĩ nhiên, nếu "tập tin" được sắp là một xâu liên kết, thì bài toán không đặt ra, vì "chỗ trống phụ trợ" cần sử dụng (cho các liên kết) dùng cho mục đích khác.

Đối với mảng, trước tiên chú ý là dễ thực hiện việc trộn M với N bằng cách chỉ cần dùng thêm chỗ trống phụ trợ cho mảng nhỏ hơn trong hai mảng mà thôi (xem bài tập 2). Điều này làm giảm yêu cầu về chỗ trống cho Mergesort đi một nửa. Thực sự có thể thực hiện tốt hơn nhiều và có thể trộn tại chỗ, dẫu cho không thực tế cho lắm.

Tính chất 12.3 *Sắp xếp bằng trộn là ổn định.*

Vì các cài đặt thực sự di chuyển các khoá chỉ trong khi trộn, nên chỉ cần kiểm chứng bản thân các phép trộn là ổn định. Nhưng chứng minh là tầm thường: vị trí tương đối của các khoá bằng nhau là không bị ảnh hưởng bởi tiến trình trộn.

Tính chất 12.4 *Sắp xếp bằng trộn không bị ảnh hưởng bởi thứ tự ban đầu của dữ liệu nhập.*



Hình 12.5 *Sắp xếp trộn của một chuyển vị ngẫu nhiên*

Trong cài đặt của chúng ta, việc nhập chỉ xác định thứ tự trong đó các phần tử được xử lý trong quá trình trộn, như thế tính chất trên được khẳng định là đúng. Các cài đặt khác của trộn liên quan đến việc kiểm tra tập tin thứ nhất có hết chưa có thể dẫn đến một sự khác biệt lớn hơn nào đó tùy thuộc vào dữ liệu nhập, nhưng không nhiều lắm. Số lần lặp rõ ràng chỉ phụ thuộc vào kích thước tập tin, không phụ thuộc vào nội dung của nó, và mỗi lần lặp cần N so sánh (thật ra trung bình là N-O(1), như đã trình bày ở trên). Nhưng trường hợp xấu nhất tương tự trường hợp trung bình.

Hình 12.4 cho thấy sắp xếp trộn từ dưới lên thao tác trên tập tin nguyên thủy theo thứ tự ngược. So sánh hình này với hình 8.9 cho thấy Shellsort thực hiện cùng thao tác.

Hình 12.5 trình bày một thao tác khác của Mergesort dùng phép hoán vị ngẫu nhiên để so sánh với các hình minh họa trong các chương trước. Đặc biệt, hình 12.5 có nét giống với hình 10.5: trong trường hợp này, Mergesort là "dạng chuyển đổi" của sắp bằng cơ số trực tiếp.

CÀI ĐẶT ĐƯỢC TỐI ƯU HÓA

Chúng ta đã chú ý đến vòng lặp trong của sắp xếp bằng trộn dựa trên mảng khi bàn đến các phần tử cầm canh, trong đó chúng ta biết là có thể tránh các phép kiểm tra biên của mảng ở vòng lặp trong bằng cách đảo ngược thứ tự của một trong các mảng. Điều này dẫn đến tính không hiệu quả trong các cài đặt ở trên: chuyển từ a sang b. Như chúng ta đã thấy đối với sắp xếp bằng cơ số trực tiếp ở chương 10 có thể tránh việc dời chuyển này bằng cách dùng hai bản sao: một dành để trộn từ a vào b, một bản khác để trộn từ b vào a.

Để thực hiện kết hợp hai cải tiến này, cần thay đổi sao cho việc trộn có thể đưa ra các mảng theo thứ tự tăng hay giảm dần. Trong phiên bản không đệ quy, có thể thực hiện bằng cách thay đổi lần lượt giữa dữ liệu xuất tăng và giảm dần; trong phiên bản đệ quy, ta có bốn thủ tục đệ quy: để trộn từ a(b) vào b(a) có kết quả theo thứ tự giảm hay tăng dần. Mỗi một trong hai thứ tự này sẽ rút ngắn vòng lặp bên trong của sắp bằng trộn thành một phép so sánh, một phép lưu trữ, hai phép tăng con trỏ (i hoặc j, và k) và một phép kiểm tra con trỏ. Sự tối ưu này đã cạnh tranh một cách hiệu quả với phép so sánh, phép tăng và kiểm tra, và phép hoán vị (từng phần) của Quicksort, và vòng lặp trong của Quicksort được thi hành khoảng $2\ln N \approx 1.38 \lg N$ lần, hơn Mergesort khoảng 38%.

TRỞ LẠI ĐỆ QUY

Các chương trình trong chương này, cùng với Quicksort là đặc trưng của các cài đặt thuật toán "chia để trị". Chúng ta sẽ thấy nhiều thuật toán có cấu trúc tương tự trong các chương sau, vì thế rất đáng để quan tâm một cách chi tiết vào một vài đặc tính cơ bản của các cài đặt này.

Quicksort thật sự là một thuật toán "chia để trị": trong một cài đặt đệ quy, hầu hết các công việc được thực hiện trước khi gọi đệ quy. Mặt khác, sắp bằng Mergesort đệ quy mang nhiều ý nghĩa của chia-để-trị hơn: trước tiên tập tin được chia làm hai phần, rồi mỗi phần được xử lý riêng. Mergesort thực sự xử lý lúc đầu chỉ một phần nhỏ, khi hoàn tất thì tập tin con lớn nhất sẽ được xử lý.

Quicksort bắt đầu bằng việc xử lý thật sự trên tập tin lớn nhất, và hoàn tất trên các tập tin nhỏ.

Điểm khác biệt này thể hiện rõ ràng trong các bản cài đặt không đệ quy của hai phương pháp sắp xếp. Quicksort phải duy trì một ngăn xếp, vì nó phải lưu các bài toán con đã được phân chia theo một phương thức phụ thuộc dữ liệu. Mergesort chấp nhận một phiên bản không đệ quy đơn giản vì phương thức để phân chia tập tin là độc lập với dữ liệu, vì vậy thứ tự mà nó xử lý các bài toán con có thể được sắp xếp lại để cho ra một chương trình đơn giản hơn.

Một khác biệt thực tế nữa là Mergesort có tính ổn định (nếu được cài đặt thích hợp), Quicksort thì không. Đối với Mergesort, nếu chúng ta giả thiết là các tập tin con đã được sắp ổn định, thì chúng ta chỉ cần bảo đảm là việc trộn được thực hiện theo một phương thức ổn định, có thể sắp xếp một cách dễ dàng. Nhưng đối với Quicksort, không dễ phân hoạch theo một phương thức ổn định, vì thế nó mất đi tính ổn định ngay cả trước khi đệ quy tham gia vào quá trình sắp.

Một lưu ý cuối cùng: giống như Quicksort hay bất kỳ một chương trình đệ quy nào khác, là Mergesort có thể được cải tiến bằng cách sử dụng các tập tin con nhỏ theo các cách khác nhau. Trong các phiên bản đệ quy của chương trình, điều này có thể được cài đặt một cách chính xác như với Quicksort, hoặc là dùng các tập tin con nhỏ sắp xếp bằng chèn hay thực hiện một lần lặp để dọn dẹp sau đó. Trong các phiên bản không đệ quy, các tập tin con nhỏ đã được sắp có thể được tạo từ một lần lặp ban đầu có sử dụng một sửa đổi thích hợp của sắp chèn hay sắp chọn. Một ý tưởng khác được đề nghị cho Mergesort là tận dụng thứ tự "tự nhiên" trong tập tin bằng cách dùng một phương pháp từ dưới lên để trộn hai đường chạy đã được sắp đầu tiên trong tập tin, sau đó tới hai đường chạy kế, ..., lặp lại tiến trình này cho đến khi tập tin đã sắp xong. Phương pháp này có vẻ rất quyến rũ, nó không đối nghịch với phương pháp chuẩn mà chúng ta đã bàn vì chi phí xác định các đường chạy nằm ở vòng lặp trong, là tiết kiệm hơn ngoại trừ các trường hợp suy thoái đặc biệt (chẳng hạn tập tin đã sắp sẵn rồi).

BÀI TẬP

1. Cài đặt một phương pháp sắp bằng trộn (Mergesort) đệ quy bằng cách cắt xén bớt phương pháp sắp chèn cho các tập tin con nhỏ hơn M phần tử; hãy xác định theo kinh nghiệm giá trị của M để nó chạy nhanh nhất trên một tập tin ngẫu nhiên gồm 1000 phần tử.

2. So sánh theo kinh nghiệm phép sắp Mergesort đệ quy và không đệ quy cho các xâu liên kết và N=1000.

3. Cài đặt phép sắp Mergesort đệ quy cho một mảng gồm N số nguyên, sử dụng một mảng phụ trợ có kích thước nhỏ hơn N/2.

4. Đúng hay sai: thời gian chạy của Mergesort không phụ thuộc vào giá trị của các khoá trong tập tin nhập. Hãy giải thích câu trả lời của bạn.

5. Số bước tối thiểu mà Mergesort có thể dùng là bao nhiêu (sai khác một hệ số hằng) ?

6. Cài đặt phép sắp Mergesort không đệ quy từ dưới lên sử dụng hai mảng thay vì dùng xâu liên kết.

7. Hãy chỉ ra các phép trộn được thực hiện khi dùng Mergesort đệ quy để sắp các khoá E A S Y Q U E S T I O N.

8. Hãy cho biết nội dung của xâu liên kết ở mỗi bước lặp khi dùng Mergesort không đệ quy để sắp các khoá E A S Y Q U E S T I O N.

9. Hãy thử thực hiện một phép Mergesort đệ quy, sử dụng mảng, lấy ý tưởng thực hiện các phép trộn 3-đường (3-way) thay vì trộn 2-đường ?

10. Hãy kiểm tra theo kinh nghiệm yêu cầu trong bài học đối với các tập tin ngẫu nhiên có kích thước 1000 : ý tưởng tận dụng thứ tự "tự nhiên" trong tập tin là không mang lại kết quả.

13
SẮP XẾP NGOÀI

Nhiều ứng dụng sắp xếp quan trọng liên quan đến việc xử lý các tập tin có kích thước lớn, nên không đủ bộ nhớ cho những tập tin quá lớn như vậy. Các phương pháp thích hợp cho các ứng dụng như vậy gọi là phương pháp sắp xếp ngoài, vì chúng liên quan đến xử lý dữ liệu bên ngoài đơn vị xử lý trung ương.

Có hai yếu tố chính khác biệt giữa thuật toán xử lý ngoài với các thuật mà chúng ta đã thấy. Trước tiên, chi phí truy xuất một phần tử trong bộ nhớ ngoài thì lớn hơn bất kỳ chi phí tính toán nào. Thứ hai là phải chịu các giới hạn nghiêm ngặt về truy xuất, phụ thuộc vào môi trường lưu trữ bên ngoài: ví dụ như các phần tử trên băng từ chỉ truy xuất theo phương thức tuần tự.

Sự khác biệt lớn của loại và chi phí thiết bị lưu trữ bên ngoài tạo sự phát triển của các phương pháp sắp xếp bên ngoài rất độc lập về kỹ thuật. Các phương pháp này có thể phức tạp và nhiều tham số ảnh hưởng đến tính năng của chúng: một phương pháp tốt có thể không được trưng dụng vì chỉ thay đổi chút đỉnh về kỹ thuật mà không có lợi cho sắp xếp ngoài. Vì lý do đó, ở chương này chúng ta sẽ tập trung vào các phương pháp tổng quát thay vì chú ý vào các cài đặt cụ thể.

Tóm lại, đối với sắp xếp ngoài, phương diện "hệ thống" chắc chắn quan trọng như phương diện "thuật toán". Cả hai lĩnh vực nên xem xét kỹ lưỡng một khi cần phát triển phương pháp sắp xếp ngoài cho hiệu quả. Chi phí cơ bản trong việc sắp xếp ngoài là dành cho nhập-xuất. Một bài tập hay dành cho người nào định cài đặt một chương trình hiệu quả để sắp xếp một tập tin có kích thước lớn trước tiên là cài đặt một chương trình hiệu quả để sao chép tập tin này, kế đó cài đặt một chương trình để đảo ngược thứ tự của các phần tử trong tập tin này. Vấn đề hệ thống nảy sinh khi thử

giải quyết các bài toán này tương tự như vấn đề hệ thống nảy sinh khi sắp xếp ngoài. Hoán vị một tập tin có kích thước lớn cũng khó như sắp xếp nó dù cho không cần phải so sánh khóa ... Trong sắp xếp ngoài, chúng ta quan tâm chủ yếu đến giới hạn thời gian di chuyển mỗi thành phần dữ liệu giữa môi trường lưu trữ bên ngoài và bộ nhớ, và bảo đảm là những lần chuyển giao như thế phải thực hiện một cách hiệu quả với khả năng của phần cứng.

Các phương pháp sắp xếp ngoài phát triển thích hợp với các thẻ đục lỗ và băng giấy trước đây, hiện tại là dùng băng từ và đĩa, các kỹ thuật nổi như bộ nhớ nổi và đĩa video. Sự khác nhau chủ yếu giữa các thiết bị là kích thước tương đối và tốc độ lưu trữ có sẵn có và kiểu giới hạn truy xuất dữ liệu. Chúng ta sẽ tập trung vào các phương pháp cơ bản để sắp xếp trên băng từ và đĩa bởi vì các thiết bị này tồn tại để sử sử dụng phổ biến và đáp ứng được hai cơ chế truy xuất khác nhau đặc trưng cho nhiều hệ thống xử lý bên ngoài. Thông thường các hệ máy tính hiện đại có "phân cấp lưu trữ" của bộ nhớ lớn hơn, rẻ hơn, chậm hơn. Nhiều thuật toán mà chúng ta sẽ tìm hiểu có thể áp dụng tốt trong môi trường như thế, nhưng chúng ta sẽ làm việc với độ phân cấp "hai mức" gồm bộ nhớ chính và đĩa hay băng từ.

SẮP XẾP TRỘN

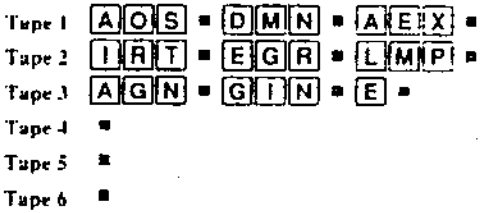
Phần lớn các phương pháp sắp xếp ngoài sử dụng chiến lược tổng quát sau: tạo một lần duyệt trên tập tin cần sắp, chia nó thành các khối có kích thước phù hợp với bộ nhớ trong, và sắp xếp các khối này. Sau đó trộn các khối đã sắp này với nhau bằng cách duyệt vài lần trên tập tin, rồi lại tạo các khối được sắp lớn hơn cho đến khi sắp xong toàn bộ tập tin. Đa số dữ liệu thường được truy xuất theo kiểu tuần tự, đây là tính chất làm cho phương pháp này thích hợp đối với các thiết bị bên ngoài. Các thuật toán sắp ngoài cố gắng rút ngắn số lần duyệt trên tập tin và giảm chi phí sao cho gần với chi phí của một bản sao chép.

Bởi vì phần lớn chi phí của phương pháp sắp xếp ngoài là dành cho nhập-xuất, nên chúng ta có thể đo lường bằng chi phí của một phương pháp trộn bằng cách đếm số lần đọc hay ghi mỗi từ trong một tập tin (số lần lặp trên toàn bộ dữ liệu). Trong nhiều ứng

dụng, các phương pháp mà chúng ta xem xét liên quan đến thứ tự của mười hay ít hơn các bước như thế. Chú ý là chúng ta quan tâm các phương pháp loại trừ một bước đơn lẻ. Thời gian thực hiện toàn bộ quá trình sắp xếp có thể đánh giá dễ dàng như thời gian thực hiện của bài tập "sao chép ngược tập tin" như đã đề cập ở trên.

TRỘN NHIỀU ĐƯỜNG CÂN BẰNG

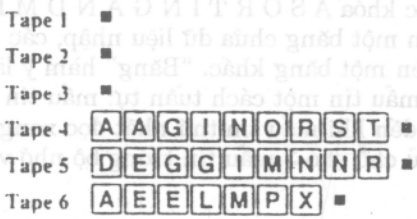
Nhờ vào một ví dụ nhỏ, chúng ta lần theo từng bước của thủ tục sắp bằng phương pháp trộn đơn giản nhất. Giả sử là chúng ta có các mẩu tin có các khóa A S O R T I N G A N D M E R G I N G E X A M P L E trên một băng chứa dữ liệu nhập, các khóa này được sắp xếp và xuất lên một băng khác. "Băng" hàm ý là chúng ta giới hạn việc đọc các mẩu tin một cách tuần tự: mẩu tin thứ hai không thể được đọc cho đến khi mẩu tin thứ nhất đọc xong ... Giả thiết là chúng ta chỉ có đủ chỗ cho 3 mẩu tin trong bộ nhớ và có sẵn nhiều băng từ.



Hình 13.1 *Trộn ba đường cân bằng: kết quả của bước thứ nhất*

Bước đầu tiên đọc ba mẩu tin một lúc, sắp xếp chúng để tạo thành khối ba mẩu tin, và xuất các khối được sắp. Vì vậy, trước tiên chúng ta đọc A S O và xuất các khối gồm A O S, kế tiếp đọc R T I và xuất khối I R T và cứ tiếp tục như thế. Theo thứ tự của các khối đã trộn xong này, chúng ta nên đặt các khối trên các băng khác nhau. Nếu chúng ta thực hiện việc trộn 3 đường, thì sử dụng ba băng, gói gọn trong các lần sắp xếp với cấu hình trình bày ở hình 13.1.

Bây giờ chúng ta sẽ sắp các khối có kích thước ba. Chúng ta đọc mẩu tin đầu tiên của mỗi băng (có đủ bộ nhớ) và lấy một mẩu tin có khóa nhỏ nhất. Kế tiếp đọc mẩu tin kế trên băng này và lấy tiếp một mẩu tin có khóa nhỏ nhất. Khi đến cuối khối có ba từ, băng tương ứng sẽ không thao tác nữa cho đến khi xử lý xong các khối của hai băng khác và chín mẩu tin được xuất ra. Sau đó quá trình được lặp lại để trộn hai hay ba từ trên mỗi băng thành một khối chín từ (xuất trên một băng khác để chuẩn bị cho lần trộn kế tiếp). Tiếp tục bằng cách này, chúng ta có ba khối dài như hình 13.2.



Hình 13.2 *Trộn ba đường cân bằng: kết quả của bước thứ hai*

Bây giờ ta thêm một bước trộn ba-đường nữa để hoàn tất việc sắp xếp. Nếu chúng ta có một tập tin dài hơn có nhiều khối với kích thước là chín trên mỗi băng, thì chúng ta sẽ hoàn tất bước thứ hai với khối có kích thước 27 trên băng 1,2,3, kế đến bước thứ ba sẽ cho ra khối có kích thước 81 trên băng 4,5,6 và cứ thế mà tiếp tục. Chúng ta cần sáu băng để sắp một tập tin lớn tùy ý: ba để nhập và ba để xuất cho mỗi lần trộn ba-đường (Thực sự chúng ta chỉ cần bốn băng: có thể chỉ xuất trên một băng, và sau đó các khối từ băng này được phân phối cho các băng giữa các lần trộn.)

Phương pháp này gọi là **trộn nhiều đường cân bằng**: một thuật toán hợp lý để sắp ngoài và cũng là một quan điểm tốt để cài đặt một phương pháp sắp ngoài. Các thuật toán phức tạp hơn nữa ở dưới đây có thể tạo đường chạy sắp xếp nhanh hơn chút nữa, nhưng không nhiều lắm. (Tuy nhiên, khi thời gian thực hiện tính bằng đơn vị giờ thì ngay cả giảm bớt một tỉ lệ nhỏ cũng đã rất đáng kể.)

Giả sử chúng ta có N từ thao tác trong quá trình sắp và bộ nhớ trong có kích thước M. Sau đó lần "sắp" cho ra khoảng N/M khối đã sắp (đánh giá này giả thiết trên các mẩu tin 1 từ: đối với mẩu tin lớn hơn: số khối được sắp được tính bằng cách nhân với kích thước mẩu tin). Nếu chúng ta thực hiện trộn P đường trong mỗi bước, thì số lần lặp tính khoảng $\log_P(N/M)$, bởi vì mỗi bước rút ngắn số khối đã sắp bởi một hệ số P.

Mặc dù với các ví dụ nhỏ có thể giúp chúng ta hiểu chi tiết thuật toán, nhưng tốt nhất là sử dụng các tập tin có kích thước lớn hơn khi làm việc với sắp ngoài. Chẳng hạn, công thức ở trên cho thấy trộn bốn-đường để sắp một tập tin 200-triệu-từ trên máy tính có một triệu từ bộ nhớ thì chỉ cần tổng cộng khoảng năm bước. Một đánh giá khác về thời gian chạy cũng tính được bằng năm lần thời gian thực hiện cho cài đặt của bản sao tập tin theo thứ tự ngược ở trên.

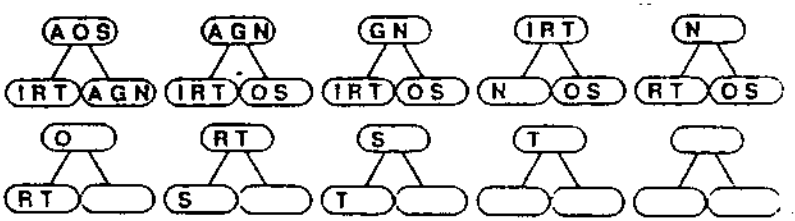
PHÉP CHỌN THAY THẾ

Vấn đề chi tiết của cài đặt cũng được phát triển bằng cách sử dụng hiệu quả hàng đợi có độ ưu tiên. Trước tiên, chúng ta nhận định là hàng đợi có độ ưu tiên đưa ra một cách thức tự nhiên để cài đặt trộn nhiều đường. Quan trọng hơn nữa, là chúng ta có thể sử dụng hàng đợi có độ ưu tiên cho lần sắp ban đầu, chúng ta có thể tạo ra các khối được sắp dài hơn lượng bộ nhớ trong cần thiết lưu trữ.

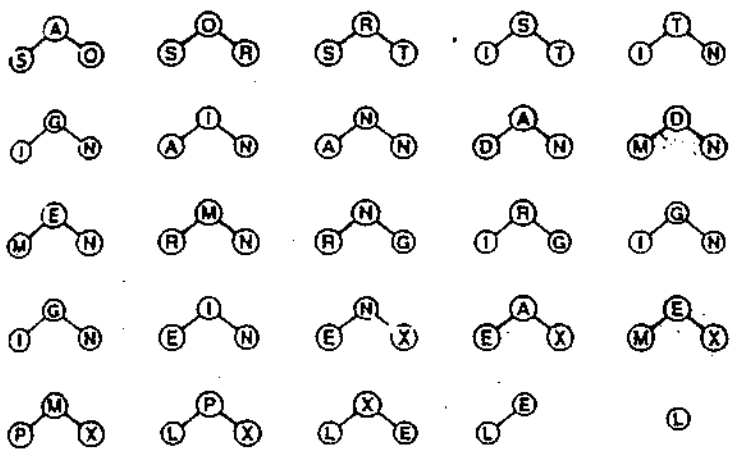
Thao tác cơ bản để trộn P-đường là lần lượt xuất ra phần tử nhỏ nhất trong các phần tử còn lại trong mỗi khối của P khối được trộn. Phần tử nhỏ nhất cần được thay thế bằng phần tử kế trong khối. Thao tác thay thế trên hàng đợi có độ ưu tiên có kích thước P là thao tác cần phải có (Thực sự, các ấn bản "gián tiếp" của chương trình hàng đợi có độ ưu tiên mô tả ở chương 11 thích hợp hơn cho ứng dụng này). Đặc biệt, để trộn P đường chúng ta bắt đầu bằng cách lấp đầy hàng đợi có độ ưu tiên kích thước P bằng phần tử nhỏ nhất của mỗi khối trong P khối, dùng thủ tục pqinssert từ chương 11 (được sửa đổi thích hợp sao cho phần tử nhỏ nhất nằm ở đỉnh của heap). Sau đó, sử dụng thủ tục pqreplace từ chương 11 (sửa đổi tương tự) chúng ta sẽ lấy ra được phần tử nhỏ nhất và thay thế nó trong hàng đợi có độ ưu tiên bằng phần tử kế của khối

Quá trình trộn A O S với I R T và A G N (lần trộn thứ nhất ở ví dụ trên) sử dụng một heap kích thước 3 như trong hình 13.3. Các "khóa" trong các heap này là khóa nhỏ nhất (khóa đầu tiên) ở mỗi nút. Rõ hơn, chúng ta trình bày toàn bộ khối trong các nút của heap; dĩ nhiên cài đặt thực sự sẽ là heap gián tiếp chứa các con trỏ đến các khối. Trước tiên, A là dữ liệu xuất để cho O (khóa kế tiếp trong khối của nó) trở thành "khóa" của gốc. Điều này vi phạm tính chất của heap, cho nên hoán vị nút này với nút chứa A, G, và N. Sau đó lấy A ra và thay thế bằng khóa kế tiếp trong khối của nó là G. Điều này không vi phạm tính chất của heap, nên không có thay đổi gì cả. Tiếp tục bằng cách này, chúng ta cho ra tập tin đã sắp (đọc khóa nhỏ nhất trong nút gốc của cây ở hình 13.3 để thấy các khóa theo thứ tự xuất hiện ở vị trí đầu tiên trên heap và các khóa này là khóa xuất). Khi hết một khối, một biến cầm canh được đặt trên heap và xem như lớn hơn tất cả các khóa khác. Khi heap chứa tất cả các biến cầm canh thì quá trình trộn hoàn tất. Phương pháp sử dụng hàng đợi có độ ưu tiên nầy đôi khi được gọi là phép chọn thay thế.

Để trộn P-đường, chúng ta có thể sử dụng phép chọn thay thế trên hàng đợi có độ ưu tiên kích thước P để lấy từng phần tử ra sau logP bước thực hiện. Sự khác nhau này không có ý nghĩa thực tiễn, bởi vì một cài đặt theo hình thức "ép buộc thô thiển" cũng có thể lấy ra từng phần tử sau P bước và thông thường P rất nhỏ đến nỗi chi phí này không đáng kể so với chi phí của việc lấy ra từng phần tử thực sự. Tầm quan trọng thực sự của phép chọn thay thế là cách thức phương pháp này được sử dụng trong phần đầu của quá trình trộn: để tạo các khối được sắp ban đầu mà cung cấp phần cơ sở cho các bước trộn.



Hình 13.3 *Giai đoạn trộn của phương pháp chọn thay thế, với heap có kích thước ba*



Hình 13.4 *Phát sinh các đường chạy ban đầu của phương pháp chọn thay thế*

Tư tưởng là truyền phần dữ liệu nhập (chưa có thứ tự) qua hàng đợi độ ưu tiên có kích thước lớn, ghi phần tử nhỏ nhất lên hàng đợi này như trên, và thay thế nó với phần tử kế của dữ liệu nhập theo qui ước sau: nếu phần tử mới nhỏ hơn phần tử xuất cuối, nó được đánh dấu như một thành phần của khối kế (vì nó không thể trở thành một thành phần của khối được sắp hiện hành) và xem như lớn hơn tất cả các phần tử trong khối hiện hành. Khi một phần tử đã bị đánh dấu ở đỉnh của hàng đợi có độ ưu tiên khối cũ chấm dứt và khối mới bắt đầu. Điều này dễ cài đặt bằng pquisert và pqreplace từ chương 11, được hiệu chỉnh thích hợp sao cho phần tử nhỏ nhất ở đỉnh heap và cài đặt bằng pqreplace được thay đổi để dùng các phần tử bị đánh dấu luôn lớn hơn các phần tử không bị đánh dấu.

Tập tin ví dụ chứng tỏ giá trị của phương pháp chọn thay thế. Với dung lượng bộ nhớ có khả năng chứa ba mẩu tin, có thể cho ra các khối được sắp có kích thước 5, 3, 6, 4, 5 và 2 như trong hình 13.4. Như trước đây, các khóa chiếm giữ vị trí đầu heap theo thứ tự giống với thứ tự xuất ra. Điều này chứng tỏ các khóa trong heap thuộc vào khối nào: một phần tử được đánh dấu cùng một cách

thức như phần tử gốc tùy thuộc vào khối được sắp hiện hành và các phần tử khác phụ thuộc vào khối được sắp xếp kế tiếp. Tính chất heap (khóa thứ nhất lớn hơn khóa thứ hai và thứ ba) luôn được bảo đảm, các phần tử trong khối được sắp xếp kế tiếp xem như lớn hơn các phần tử trong khối được sắp hiện hành. Đường chạy đầu tiên chấm dứt bằng I N G trong heap, vì các khóa này đi kèm với các khóa lớn hơn ở gốc (vì thế chúng không thể gộp vào đường chạy đầu tiên), đường chạy thứ hai kết thúc bằng A D N trên heap.

TÍNH CHẤT 13.1. *Đối với các khóa ngẫu nhiên các đường chạy sản sinh từ phép chọn thay thế bằng khoảng hai lần kích thước của heap.*

Chứng minh tính chất này thực sự cần phân tích phức tạp, nhưng dễ kiểm chứng bằng thực nghiệm.

Hiệu quả của đường chạy này là tiết kiệm một lần trộn: thay vì bắt đầu bằng các đường chạy đã sắp có kích thước bộ nhớ trong và thực hiện một lần trộn để cho ra các đường chạy khoảng hai lần kích thước bộ nhớ trong, chúng ta có thể tạo các đường chạy như vậy bằng cách sử dụng chọn thay thế bằng một hàng đợi có độ ưu tiên kích thước M. Nếu có dùng các khóa có thứ tự, thì các đường chạy sẽ dài hơn nhiều. Chẳng hạn, nếu không có khóa nào có nhiều hơn M khóa lớn hơn phía trước nó trong tập tin, tập tin sẽ được sắp hoàn toàn bằng lần chọn thay thế và không cần lần trộn nào. Đây là lý do thực tế quan trọng nhất để sử dụng phương pháp này.

Tóm lại, kỹ thuật chọn thay thế có thể sử dụng cho cả hai "sắp xếp" và "trộn" của trộn nhiều đường cân bằng.

Tính chất 13.2: *Một tập tin gồm N mẩu tin có thể được sắp hàng bằng cách sử dụng bộ nhớ trong có khả năng lưu trữ N mẩu tin và P+1 băng tốn khoảng 1+log (N/2M) lần lặp.*

Như đã trình bày ở trên, trước tiên chúng ta sử dụng phép chọn thay thế bằng hàng đợi có độ ưu tiên có kích thước M để cho ra các đường chạy ban đầu có kích thước khoảng 2M (trong trường hợp ngẫu nhiên) hay nhiều hơn (nếu tập tin đã sắp từng phần), sau đó sử dụng phép chọn thay thế có hàng đợi có độ ưu tiên có kích thước P cần khoảng log (N/2M) (hay ít hơn) lần trộn.

CÁC ĐỀ NGHỊ THỰC TẾ

Để hoàn tất cài đặt phương pháp sắp xếp ở trên, cần thiết cài đặt các hàm nhập xuất thực sự chuyển đổi dữ liệu bộ xử lý và các thiết bị bên ngoài. Các hàm này dĩ nhiên là phương tiện thực hiện tốt để sắp xếp ngoài, và chúng cần cân nhắc đối với một vài hệ thống.

Mục tiêu chính trong cài đặt là gối nhau các phần đọc, ghi, tính toán càng nhiều càng tốt. Phần lớn hệ thống máy tính lớn có đơn vị xử lý độc lập để điều khiển các thiết bị nhập xuất tỷ lệ lớn thỏa điều kiện trên. Hiệu quả thực hiện của phương pháp sắp ngoài phụ thuộc vào số thiết bị sẵn có.

Đối với mỗi tập tin được đọc hay ghi, kỹ thuật lập trình hệ thống chuẩn gọi là "hai vùng đệm" (double-buffering) sử dụng để nâng tối đa khả năng gối chồng của nhập xuất với việc tính toán. Ý tưởng là quản lý hai "vùng đệm", một để cho bộ xử lý, và một cho thiết bị nhập xuất (hay bộ xử lý dùng để điều khiển thiết bị nhập xuất). Để nhập, bộ xử lý dùng một vùng đệm của nó và sau đó các vùng đệm chuyển vai trò : bộ xử lý sử dụng dữ liệu mới trong vùng đệm bằng dữ liệu do bộ xử lý và thiết bị dành riêng. Thông thường thời gian nhập xuất lớn hơn thời gian xử lý và hiệu quả của kỹ thuật "hai vùng đệm" là gối chồng thời gian tính toán, vì vậy vùng đệm càng lớn càng tốt.

Khó khăn của hai vùng đệm là thực sự chỉ sử dụng khoảng nửa chỗ trống bộ nhớ có sẵn. Điều này có thể dẫn đến tính không hiệu quả nếu có nhiều vùng đệm, như là trường hợp trộn P đường khi P không nhỏ. Vấn đề này có thể sử dụng kỹ thuật gọi là "dự báo", kỹ thuật này chỉ cần sử dụng một vùng đệm trong quá trình trộn. Kỹ thuật này thực hiện như sau. Chắc chắn cách tốt nhất để gối chồng việc nhập với tính toán trong quá trình thay thế là gối chồng việc nhập của vùng đệm cần lấp đầy kế tiếp bằng phần đang xử lý của thuật toán. Dễ dàng xác định đây là vùng đệm nào : vùng đệm nhập kế tiếp được làm rỗng là vùng đệm chứa phần tử cuối là phần tử nhỏ nhất. Chẳng hạn, khi trộn A O S với I R T và A G N chúng ta biết là vùng đệm thứ 3 sẽ là vùng đệm làm rỗng đầu tiên, rồi đến vùng đệm thứ 1. Một cách đơn giản để gối chồng xử lý bằng dữ liệu nhập của trộn nhiều đường là giữ thêm một vùng đệm lấp

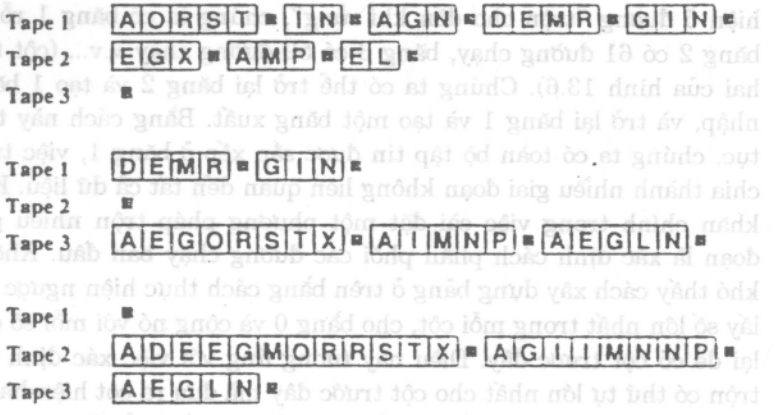
đầy bởi thiết bị nhập tùy theo luật này. Khi bộ xử lý gặp một vùng đệm rỗng nó đợi cho đến khi lấy đầy vùng đệm nhập, rồi chuyển sang dùng vùng đệm đó và điều khiển thiết bị nhập để lấp đầy vùng đệm , vùng đệm này chỉ rỗng tùy theo luật "dự báo" này.

Vấn đề quan trọng trong cài đặt của trộn nhiều đường là chọn lựa giá trị của P, "thứ tự" của việc trộn. Đối với sắp trên băng, chỉ khi truy xuất tuần tự, sự lựa chọn này rất dễ : P phải ít hơn số băng một giá trị bởi vì trộn nhiều đường dùng P băng nhập và một băng xuất. Dĩ nhiên có ít nhất hai băng nhập, vì thế không cần thử sắp xếp trên băng với ít hơn ba băng.

Đối với sắp xếp , khi truy xuất đến vị trí tùy ý thì được quyền nhưng tốn hơn truy xuất tuần tự, cũng hợp lý để chọn P ít hơn số đĩa sẵn có là một để tránh chi phí lớn của truy xuất không tuần tự, chẳng hạn nếu hai tập tin dữ liệu nhập khác nhau trên cùng một đĩa. Thông thường sử dụng khác nữa là chọn P đủ lớn để sắp xếp hoàn tất trong hai giai đoạn trộn : không hợp lý để thử sắp xếp bằng một lần trộn, nhưng sắp bằng hai lần trộn có thể thực hiện với P khá bé. Bởi vì phương pháp chọn thay thế cho ra khoảng N/2M đường chạy và mỗi lần trộn chia số đường chạy cho P, điều này nghĩa là P nên chọn là số nguyên nhỏ với P<N/2M Đối với việc sắp xếp một tập tin 200 triệu từ trên máy tính với 1 triệu từ bộ nhớ, điều này hàm ý là P = 11 sẽ là cách chọn an toàn để đảm bảo sắp hai lần. (giá trị chính xác nhất của P phụ thuộc vào nhiều tham số hệ thống)

TRỘN NHIỀU LẦN

Một vấn đề trộn nhiều đường cân bằng để sắp bằng băng là nó cần hoặc là một số băng hay bản sao. Đối với trộn P đường chúng ta cũng dùng hai 2P băng (P để nhập, P để xuất) hay phải sao chép hầu hết tập tin từ một băng xuất duy nhất vào P băng chứa dữ liệu nhập giữa các lần trộn tính ra gấp đôi số lần khoảng 2 log (N/2M). Một vài thuật toán sắp xếp băng hiệu quả đã khám phá ra rằng hủy bỏ các bảng sao bằng cách dùng trộn các khối nhỏ đã được sắp. Phương pháp này gọi là trộn nhiều giai đoạn. Ý tưởng cơ bản của phương pháp này là phân phối các khối đã sắp bằng phương pháp chọn thay thế rồi áp dụng chiến thuật "trộn cho đến khi rỗng",



Hình 13.5 *Các trạng thái khởi động của trộn nhiều lần với ba băng*

theo quan điểm là một trong các băng nhập và xuất thay đổi vai trò cho nhau.

Chẳng hạn, giả thiết là chúng ta chỉ có ba băng, và chúng ta bắt đầu hình ban đầu của các khối được sắp trên băng trình bày ở hình 13.5 (Điều này xuất phát từ việc áp dụng chọn thay thế cho tập tin ví dụ có bộ nhớ trong có thể chỉ lưu trữ được hai mẫu tin). Băng ba được khởi tạo rỗng lúc đầu, băng xuất dành cho lần trộn đầu. Sau ba lần trộn hai đường từ băng 1 và 2 vào 3, băng thứ hai trở thành rỗng như ở giữa hình 13.5. Sau đó, hai lần trộn hai đường từ băng 1 và 3 vào 2, băng thứ 1 rỗng, xem cuối hình 13.5. Sắp xếp hoàn tất sau hai bước nữa. Trước tiên, lần trộn hai đường từ băng 2 và 3 vào 1 để lại một tập tin trên băng m2, một tập tin trên băng 1. Sau đó 1 lần lượt trộn 2 đường băng 1 và 2 vào 3 cho ra một tập tin đã sắp toàn bộ ở băng 3.

Chiến lược trộn cho đến khi rỗng này có thể mở rộng cho số băng tùy ý. Hình 13.6 cho thấy 6 băng được dùng để sắp 497 đường chạy ban đầu. Nếu ta bắt đầu bằng băng 2 (cột đầu của hình 13.6) là băng xuất, băng 1 có 61 đường chạy ban đầu, băng 3 có

120 đường chạy ban đầu v.v...(cột đầu của hình 13.6); sau đó thực hiện 5 đường "trộn cho đến khi rỗng", chúng ta có băng 1 rỗng, băng 2 có 61 đường chạy, băng 3 có 59 đường chạy v.v... (cột thứ hai của hình 13.6). Chúng ta có thể trở lại băng 2 và tạo 1 băng nhập, và trở lại băng 1 và tạo một băng xuất. Bằng cách này tiếp tục, chúng ta có toàn bộ tập tin được sắp xếp ở băng 1, việc trộn chia thành nhiều giai đoạn không liên quan đến tất cả dữ liệu. Khó khăn chính trong việc cài đặt một phương pháp trộn nhiều giai đoạn là xác định cách phân phối các đường chạy ban đầu. Không khó thấy cách xây dựng bảng ở trên bằng cách thực hiện ngược lại: lấy số lớn nhất trong mỗi cột, cho bằng 0 và cộng nó với mỗi số còn lại để có cột trước đây. Điều này tương ứng với việc xác định lần trộn có thứ tự lớn nhất cho cột trước đây mà đưa ra cột hiện hành. Kỹ thuật này thực hiện trên số băng bất kỳ (tối thiểu 3) : các con số tăng theo dãy số Fibonaci tổng quát có nhiều tính chất rất hay.

Dĩ nhiên, số đường chạy ban đầu không biết được khi cải tiến, và có lẽ chính xác nó sẽ không là số Fibonaci. Vì vậy một số đường chạy "giả" nên thêm vào để tạo số đường chạy ban đầu chính xác cần cho bảng.

Phân tích của trộn nhiều giai đoạn rất phức tạp và lý thú và đưa ra các kết quả ngạc nhiên. Ví dụ như, phương pháp rất tốt để phân phối các đường chạy giả trong số các băng liên quan đến việc sử dụng thêm các giai đoạn và cần nhiều đường chạy giả. Lý do là vì vài đường chạy dùng trong các lần trộn nhiều hơn.

Nhiều yếu tố khác cần phải lưu ý đến trong khi cài đặt một phương pháp sắp bằng băng hiệu quả nhất. Một yếu tố chính mà chúng ta không cần quan tâm gì cả là thời gian nó cần để trở lại các băng. Vấn đề này đã được nghiên cứu và người ta đã định nghĩa nhiều phương pháp lý thú. Tuy nhiên, như đã đề cập ở trên, các tiết kiệm lưu trữ thực hiện qua trộn nhiều đường cân bằng đơn giản rất giới hạn. Ngay cả trộn nhiều giai đoạn tốt hơn trộn cân bằng chỉ với P nhỏ. Với P<=8, trộn cân bằng có thể chạy nhanh hơn trộn nhiều giai đoạn, và với P nhỏ hơn hiệu quả của trộn nhiều giai đoạn cơ bản lưu giữ hai băng (trộn cân bằng có thêm hai băng sẽ chạy nhanh hơn)

Tape 1	61	0	31	15	7	3	1	0	1
Tape 2	0	61	30	14	6	2	0	1	0
Tape 3	120	59	28	12	4	0	2	1	0
Tape 4	116	55	24	8	0	4	2	1	0
Tape 5	108	47	16	0	8	4	2	1	0
Tape 6	92	31	0	16	8	4	2	1	0

Hình 13.6 *Phân bố đường chạy cho việc trộn nhiều lần 6 băng*

MỘT CÁCH DỄ HƠN

Nhiều hệ máy tính hiện đại cung cấp khả năng bộ nhớ ảo lớn mà không nên lướt qua trong phương pháp cài đặt để sắp các tập tin lớn. Trong hệ thống có bộ nhớ ảo tốt, lập trình viên có thể sử dụng một lượng dữ liệu rất lớn, trang bị khả năng chuyển giao dữ liệu từ bộ lưu trữ bên ngoài vào bên trong khi cần. Chiến lược này dựa vào tính chất là nhiều chương trình có tham chiếu cục bộ tương đối nhỏ : mỗi tham chiếu đến bộ nhớ có lẽ là một vùng bộ nhớ gần với vùng tham chiếu khác. Điều này hàm ý các chuyển giao từ bộ nhớ lưu trữ bên ngoài vào bên trong không thường xuyên. Một phương pháp sắp xếp trong có tham chiếu cục bộ có thể thực hiện rất tốt trên hệ bộ nhớ ảo (chẳng hạn, Quicksort có hai tính "cục bộ" : phần lớn tham chiếu gần với một trong hai con trỏ phân hoạch. Nhưng kiểm tra với người lập trình hệ thống trước khi mong muốn tiết kiệm đáng kể : phương pháp sắp bằng cơ số không có tham chiếu cục bộ sẽ là một trở ngại về hệ thống bộ nhớ ảo và ngay cả Quicksort sẽ có các vấn đề phụ thuộc vào hệ thống bộ nhớ ảo sẵn có được cài đặt như thế nào. Mặt khác, chiến lược sử dụng phương pháp sắp xếp để sắp các tập tin đĩa cần thận trọng trong môi trường bộ nhớ ảo tốt.

TÀI LIỆU VỀ SẮP XẾP

Tham chiếu cơ bản cho phần này là tập 3 trong dãy các tập của D.E. Knuth về sắp xếp và tìm kiếm. Thông tin về đề tài này có thể

tìm thấy trong cuốn sách này. Đặc biệt, các kết quả ở đây về đặc trưng thuật toán được lưu trữ bởi các phép phân tính toán học.

Có nhiều tài liệu về sắp xếp. Thư mục của Knuth và Rivest vào 1973 chứa hàng trăm bản, và điều này không sử dụng sắp xếp về sách và bài báo về các vấn đề khác một tham khảo hợp thời hơn với thư mục bao phủ công việc đến 1984 là sách của Gonnet.

Đối với Quicksort, tham khảo tốt nhất là nguyên bản của Hoare 1962, đưa ra tất cả các thay đổi quan trọng, bao gồm sử dụng chọn lựa ở chương 9. Nhiều chi tiết về phân tích toán học và hiệu quả thực tế của nhiều hiệu chỉnh được đưa ra qua những năm tháng tìm thấy ở sách 1978 của tác giả này.

Một ví dụ tốt của cấu trúc hàng đợi có độ ưu tiên cải tiến là J. Wuillemin là "hàng đợi nhị thức" như được cài đặt và phân tích bởi M.R. Brown. Cấu trúc dữ liệu này hỗ trợ tất cả các thao tác hàng đợi có độ ưu tiên một cách có hiệu quả. Kỹ thuật trong loại cấu trúc dữ liệu để cài đặt thực tế là "heap" do Fredman, Sedgenick, Sleator và Tarjan trình bày.

Để có tư tưởng để rút ngắn thuật toán, chúng ta đã tìm hiểu các cài đặt thực tế, người đọc nên nghiên cứu tham khảo vật chất các chương trình tiện ích để sắp xếp trong hệ thống máy tính. Chúng cần thiết xử lý các dạng của khóa, mẩu tin và tập tin cũng như các chi tiết khác, thường lý thú khi xác định các thuật toán đóng vai tròn như thế nào.

BÀI TẬP

1. Hãy mô tả làm thế nào bạn thực hiện được phép chọn ngoại (external selection) : tìm phần tử lớn nhất thứ k trong một tập tin N phần tử, trong đó N là lớn hơn nhiều kích thước của một tập tin nằm vừa khít trong bộ nhớ chính.

2. Cài đặt thuật toán chọn thay thế, sau đó dùng nó để kiểm tra yêu cầu là : các đường chạy được sinh ra là khoảng gấp hai lần kích thước bộ nhớ trong.

3. Trường hợp xấu nhất có thể xảy ra khi phép chọn thay thế được dùng để sinh ra các đường chạy khởi đầu trong một tập tin gồm N bản ghi, sử dụng một hàng ưu tiên có kích thước M với M<N

4. Bạn sắp như thế nào dữ liệu của một đĩa nếu không có bộ nhớ nào khác (ngoài bộ nhớ chính) là dùng được ?

5. Bạn sẽ sắp xếp như thế nào dữ liệu của một đĩa nếu chỉ có một băng từ (tape) (và bộ nhớ chính) là dùng được ?

6. Hãy so sánh phép trộn cân bằng nhiều đường 4-tape và 6-tape với phép trộn nhiều bước với cùng số tape, cho 31 đường chạy khởi đầu.

7. Có bao nhiêu bước mà phép trộn nhiều bước 4-tape sử dụng khi được bắt đầu với 4-tape chứa 26, 15, 22, và 28 đường chạy khởi đầu ?

8. Giả sử 31 đường chạy khởi đầu trong một phép trộn nhiều bước 4-tape mỗi cái dài một bản ghi (ban đầu được phân bố 0, 13, 11, 7) có bao nhiêu bản ghi trong mỗi một tập tin được nói đến trong phép trộn 3 đường cuối cùng ?

9. Các tập tin nhỏ sẽ được kiểm soát như thế nào trong một bản cài đặt Quicksort được thi hành trên một tập tin rất lớn trong một môi trường bộ nhớ ảo ?

10. Bạn sẽ tổ chức như thế nào một hàng ưu tiên ngoại ? (cụ thể, hãy thiết kế một cách để hỗ trợ các thao tác insert và remove trong chương 11, khi số phần tử trong hàng ưu tiên có thể phình thật lớn so với kích thước một hàng nằm khít trong bộ nhớ chính)

14
CÁC PHƯƠNG PHÁP TÌM KIẾM CƠ BẢN

Việc tìm kiếm là thao tác nền móng cho rất nhiều tác vụ tính toán, tìm kiếm nghĩa là tìm một hay nhiều mẩu thông tin từ một số lượng lớn thông tin đã được lưu trữ. Thông thường thông tin được chia thành các mẩu tin (record), mỗi mẩu tin có một KHÓA (key) dùng cho việc tìm kiếm. Mục đích của việc tìm kiếm là tìm tất cả các mẩu tin mà khóa của chúng đồng nhất với một khóa đã cho trước. Sau khi một mẩu tin đã được tìm thấy, thông tin bên trong của nó sẽ cung cấp cho một quá trình xử lý nào đó.

Việc tìm kiếm được áp dụng rất đa dạng và rộng rãi. Ví dụ, một nhà băng cần theo dõi tất cả các cân đối tài khoản của các khách hàng và cần phải tìm kiếm để kiểm tra các biến động. Một hệ thống trợ giúp bán vé máy bay cũng có các yêu cầu tương tự, tuy nhiên hầu hết các dữ liệu trong ứng dụng nầy có thời gian sống ngắn hơn.

Hai thuật ngữ thường dùng để mô tả các cấu trúc dữ liệu cho việc tìm kiếm là TỰ ĐIỂN và BẢNG KÝ HIỆU. Ví dụ, trong một tự điển tiếng Anh, "khóa" là từ và "mẩu tin" là diễn giải cho từ đó, mỗi mẩu tin chứa định nghĩa, phát âm, và các thông tin khác. Chúng ta có thể tìm hiểu và đánh giá các phương pháp tìm kiếm bằng cách suy nghĩ làm thế nào để cài đặt một hệ thống tra tự điển tiếng Anh. Bảng ký hiệu chính là tự điển cho chương trình: "các khóa" là các tên ký hiệu dùng trong chương trình, và "các mẩu tin" chứa thông tin mô tả đối tượng được đặt tên.

Trong tìm kiếm (cũng như sắp xếp) chúng ta có những chương trình được dùng thường xuyên và rộng rãi, vì vậy sẽ rất có ích khi nghiên cứu chi tiết nhiều phương pháp khác nhau. Giống như sắp

xếp, trước tiên chúng ta sẽ quan sát các một vài phương pháp sơ đẳng rất có ích khi dùng với các bảng nhỏ và một số trường hợp đặc biệt, kế đến sẽ minh họa các kỹ thuật cơ bản được dùng trong các phương pháp nâng cao. Chúng ta sẽ nghiên cứu các phương pháp lưu trữ các mẩu tin trong các mảng mà được tìm kiếm với với sự so sánh khóa hoặc được đặt chỉ mục (index) bởi các giá trị khóa. Sau đó, chúng ta sẽ xem xét một phương pháp cơ bản để xây dựng các cấu trúc được định nghĩa bởi các giá trị khóa.

Cũng như đối với các hàng đợi, cách tốt nhất để suy nghĩ về các thuật toán tìm kiếm là đưa ra các thao tác tổng quát được rút ra từ các cài đặt cụ thể, sao cho các cài đặt khác nhau có thể được thay thế dễ dàng. Các thao tác bao gồm:

INITIALIZE khởi tạo cấu trúc dữ liệu.
SEARCH tìm kiếm một hay nhiều mẩu tin có một khóa đã cho.
INSERT chèn thêm một mẩu tin mới.
JOIN nối hai tự điển để tạo thành một tự điển lớn hơn.
SORT sắp xếp tự điển; xuất ra tất cả các mẩu tin theo thứ tự được sắp xếp.

Đôi khi các thao tác nầy được tổ hợp thành một thao tác phức tạp hơn, ví dụ một thao tác SEARCH_and_INSERT (tìm kiếm và chèn vào) thường được dùng trong các trường hợp các mẩu tin với khóa bằng nhau không được phép lưu trữ trong cấu trúc dữ liệu. Trong nhiều phương pháp, cứ mỗi lần xác định một khóa nào đó không có trong cấu trúc dữ liệu thì trạng thái nội của thủ tục tìm kiếm sẽ chứa chính xác thông tin cần thiết để chèn thêm một mẩu tin mới với khóa đã cho.

Các mẩu tin với các khóa bằng nhau có thể được xử lý bởi nhiều phương pháp, phụ thuộc vào từng ứng dụng cụ thể. Trước tiên, chúng ta sẽ xét các cấu trúc dữ liệu tìm kiếm cơ sở chỉ chứa các mẩu tin với các khóa phân biệt. Sau đó mỗi "mẩu tin" trong cấu trúc dữ liệu nầy phải chứa, chẳng hạn như một xâu liên kết của tất cả các mẩu tin có cùng khóa đó. Đây là sự dàn xếp quen thuộc từ quan điểm của người thiết kế các thuật toán tìm kiếm, và sự qui ước nầy cũng được dùng trong một số những ứng dụng khi tất cả các mẩu tin với cùng một khóa được trả về bởi một thao tác

SEARCH. Khả năng thứ hai là giữ tất cả các mẩu tin với các khóa bằng nhau trong cấu trúc dữ liệu tìm kiếm cơ sở và trả về bất kỳ một mẩu tin nào nếu có trong mỗi lần SEARCH. Trường hợp nầy thì đơn giản hơn, thường xảy ra với các ứng dụng mà xử lý mỗi lần một mẩu tin và thứ tự xử lý các mẩu tin với khóa bằng nhau là không quan trọng. Một khả năng thứ ba là là giả sử rằng mỗi mẩu tin, ngoài khóa ra, sẽ có thêm một chỉ danh duy nhất và đòi hỏi rằng mỗi thao tác SEARCH tìm thấy một mẩu tin với một chỉ danh đã cho. Còn nhiều sơ đồ phức tạp hơn có thể dùng để phân biệt các mẩu tin có khóa bằng nhau.

Mỗi thao tác được liệt kê bên trên đều có những ứng dụng rất quan trọng và một số lớn những tổ chức dữ liệu cơ sở đã được đề nghị để có thể dùng phối hợp các thao tác trên một cách hiệu quả. Trong chương nầy và một vài chương tới, chúng ta sẽ quan tâm đến cách cài đặt các hàm cơ bản SEARCH và INSERT (và dĩ nhiên cũng đề cập đến INITIALIZE), cùng với một vài ghi chú về DELETE và SORT khi cần thiết. Giống như các hàng đợi, thao tác JOIN đòi hỏi các kỹ thuật nâng cao hơn mà chúng ta chưa thể nghiên cứu chúng ở đây.

TÌM KIẾM TUẦN TỰ

Phương pháp đơn giản nhất để tìm kiếm là lưu trữ các mẩu tin trong một mảng, và kế đó duyệt xuyên qua toàn bộ mảng một cách tuần tự, mỗi lần tìm thấy một mẩu tin. Đoạn chương trình ở trang sau đưa ra một cài đặt của các hàm cơ sở sử dụng phương pháp quản lý đơn giản nầy và minh họa một vài qui ước mà chúng ta sẽ dùng để cài đặt các thuật toán tìm kiếm.

Đoạn mã nầy xử lý các mẩu tin có các khóa (key) và "thông tin đi kèm" có giá trị nguyên (info). Giống như sắp xếp, nhiều ứng dụng rất cần mở rộng các chương trình để làm việc trên các mẩu tin và khóa phức tạp hơn, nhưng điều nầy sẽ không làm thay đổi nhiều các thuật toán. Ví dụ, trường info có thể thay đổi thành một con trỏ đến một cấu trúc mẩu tin phức tạp. Trong trường hợp như thế, trường nầy có thể xem như một chỉ danh duy nhất của mẩu tin để phân biệt những mẩu tin có khóa bằng nhau.

```
type node=record
                key,   info: integer;
            end;
var a: array[1..maxN] of node; N: integer;
procedure initialize;
   begin N := 0;
   end;
function seqsearch(v:integer; x:integer): integer;
   begin
      a[N+1].key := v;
      if x then repeat x:=x+1 until v=a[x].key;
      seqsearch := x;
   end;
function seqinsert(v:integer): integer;
   begin
      N := N+1; a[N].key:=v; seqinsert := N;
   end;
```

Thủ tục seqsearch có hai đối số trong cài đặt này: một giá trị khóa và một chỉ số mảng (biến x). Chỉ số này dùng để xử lý trường hợp nhiều mẩu tin có cùng một giá trị khóa: bằng cách thực hiện liên tục t:=seqsearch(v, t) khởi đầu với t=0 chúng ta có thể cho t lần lượt nhận từng chỉ số của các mẩu tin có khóa bằng v.

Một mẩu tin đặc biệt có khóa là khóa đang tìm được thêm vào; điều này bảo đảm rằng quá trình tìm kiếm sẽ kết thúc và cũng làm đơn giản hơn phép kiểm tra trong vòng lặp. Sau khi vòng lặp dừng, nếu chỉ số trả về nhỏ hơn hay bằng N thì đó là chỉ số của mẩu tin tìm được, nếu ngược lại thì trong bảng đã cho không có mẩu tin nào có giá trị khóa muốn tìm. Kỹ thuật này giống như kỹ thuật dùng một mẩu tin chứa giá trị khóa nhỏ nhất hay lớn nhất để làm đơn giản các vòng lặp của các thuật toán sắp xếp trong các chương 8-12 hơn.

TÍNH CHẤT 14.1 *Tìm kiếm tuần tự (cài đặt mảng) sử dụng đúng N+1 phép so sánh cho một lần tìm kiếm không thành công và trung bình khoảng N/2 phép so sánh cho một lần tìm kiếm thành công.*

Với trường hợp tìm kiếm không thành công, tính chất này được suy ra trực tiếp từ đoạn chương trình. Trong trường hợp tìm kiếm thành công, nếu chúng ta giả sử rằng mỗi mẩu tin có khả

năng tìm thấy là như nhau, thì số trung bình của số lần so sánh là:

$$\frac{1+2+3+\ldots N}{N} = \frac{N+1}{2}$$

con số này bằng đúng một nửa của trường hợp tìm kiếm không thành công.

Tìm kiếm tuần tự có thể được cài đặt bằng một phương pháp tự nhiên bằng cách dùng một xâu liên kết để biểu diễn các mẩu tin. Một trong những thuận lợi của cách làm làm nầy là nó có thể giữ cho xâu được sắp xếp, và nhờ đó mà có thể tìm kiếm nhanh hơn.

```
type  link = ↑node;
      node = record key, info: integer; next: link;
                    end;
var head,t,z:link; i:integer;
procedure initialize;
   begin
   new(z); z↑.next := z;
   new(head);head↑ .next := z;
   end;
function listsearch(v:integer; t:link):link;
   begin
   z↑.key := v;
   repeat
      t := t↑.next
   until v <=t↑.key;
   if v=t↑.key
      then listsearch := t
      else listsearch := z;
   end;
```

Với một xâu đã được sắp xếp, một lần tìm kiếm có thể kết thúc không thành công khi một mẩu tin có khóa lớn hơn khóa đang tìm kiếm được tìm thấy. Do đó, về mặt trung bình, chỉ khoảng một nửa các mẩu tin (không phải tất cả) cần được kiểm tra cho một lần tìm kiếm không thành công. Thứ tự sắp xếp được duy trì dễ dàng bởi vì một mẩu tin mới có thể chèn vào xâu một cách dễ dàng tại vị trí mà việc tìm kiếm kết thúc không thành công:

```
function listinsert(v:integer; t:link):link;
  var x: link;
  begin
    z↑.key := v;
    while t↑.next↑.key < v do t := t↑.next;
    new(x); x↑.next := t↑.next; t↑.next := x;
    x↑.key := v;
    listinsert := x;
  end;
```

Như thường lệ đối với các xâu liên kết, có thể một nút dẫn đầu head và một nút kết thúc z để làm đơn giản chương trình. Do đó, lệnh listinsert(v, head) sẽ đặt một nút mới với khóa v vào trong xâu được trỏ tới bởi trường next của head, và listsearch thì cũng tương tự. Các lần gọi listsearch kế tiếp dùng các xâu trả về của lần trước đó sẽ trả về các mẩu tin với các khóa bằng nhau. Nút kết thúc z được dùng cho trường hợp không thành công, nghĩa là nếu listsearch trả về z thì sự tìm kiến không thành công.

TÍNH CHẤT 14.2 *Tìm kiếm tuần tự (cài đặt bằng xâu có thứ tự) sử dụng trung bình khoảng N/2 phép so sánh cho cả hai trường hợp tìm kiếm thành công và không thành công.*

Trường hợp tìm kiếm thành công giống như đã bàn ở trên. Trong trường hợp tìm kiếm không thành công, nếu chúng ta giả sử rằng khả năng tìm kết thúc ở mỗi phần tử trong xâu và nút z là như nhau thì số trung bình của số lần so sánh là:

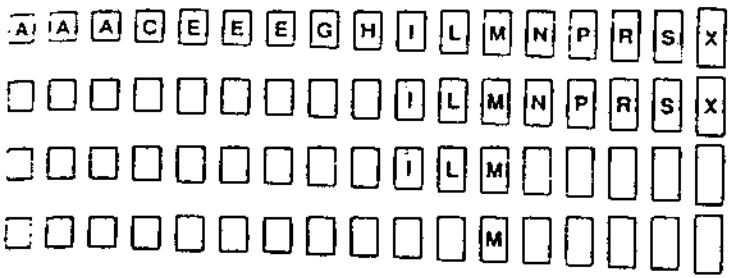
$$1+2+...+N+(N+1) = (N+2)(N+1)/2$$

Nếu được biết thông tin về tần số truy xuất các mẩu tin thì chúng ta có thể cải tiến thuật toán tìm kiếm bằng cách sắp xếp các mẩu tin một cách thông minh. Một sắp xếp "tối ưu" là đặt mẩu tin được truy xuất thường xuyên nhất ở vị trí đầu tiên, mẩu tin có tần số truy xuất thường xuyên thứ hai được đặt ở vị trí thứ hai... Kỹ thuật nầy có thể rất hiệu quả, đặc biệt trong trường hợp chỉ có một tập hợp nhỏ các mẩu tin được truy xuất thường xuyên.

Nếu thông tin về tần số truy xuất không sẵn có, thì một sự xấp xỉ đến sắp xếp tối ưu có thể làm bằng cách tìm kiếm "tự tổ chức": mỗi lần một mẩu tin được truy xuất, thì di chuyển nó lên vị trí đầu tiên của xâu. Phương pháp nầy sẽ dễ dàng cài đặt nếu dùng một

xâu liên kết. Dĩ nhiên thời gian thực hiện của phương pháp phụ thuộc vào sự phân bố của các mẩu tin, rất khó tiên đoán nó trong trường hợp tổng quát. Tuy nhiên, phương pháp nầy sẽ thích hợp trong trường hợp mỗi mẩu tin có khuynh hướng gần những mẩu tin khác.

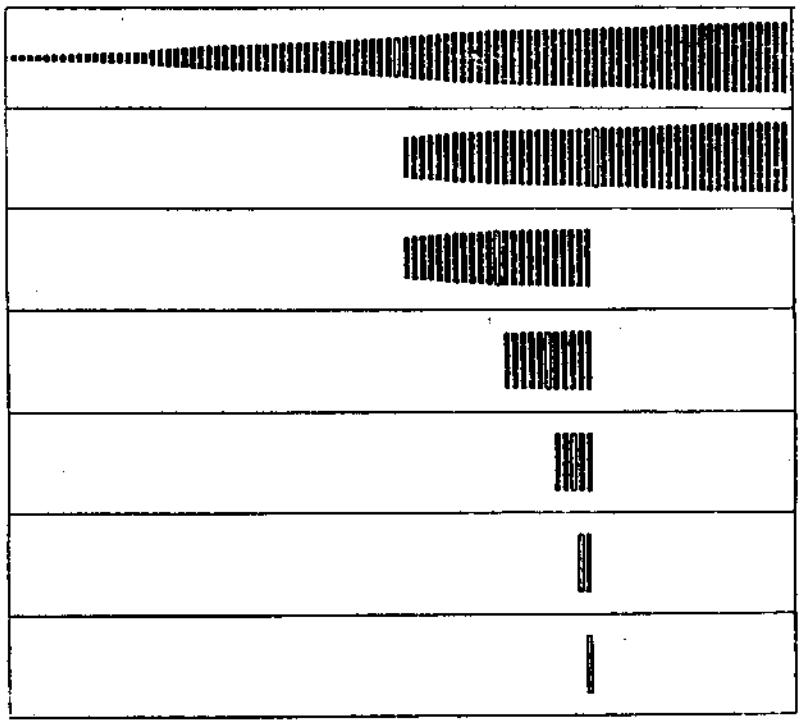
TÌM KIẾM NHỊ PHÂN



Hình 14.1 *Tìm kiếm nhị phân*

Nếu tập hợp các mẩu tin lớn thì tổng số thời gian tìm kiếm sẽ được rút ngắn bằng cách dùng một thủ tục tìm kiếm dựa trên sự ứng dụng sơ đồ "chia-để-trị": chia tập hợp các mẩu tin thành hai phần, xác định xem phần nào chứa khóa, kế đến tiếp tục quá trình cho phần chứa khóa. Sở dĩ chúng ta có thể chia đôi và chỉ tìm tiếp tục tìm trên một nửa các mẩu tin là nhờ vào giả thiết các mẩu tin được sắp xếp theo thứ tự khóa. Có thể giả sử các mẩu tin được sắp xếp tăng, trường hợp sắp xếp giảm thì tương tự. Để tìm một khóa v có trong bảng hay không, trước tiên ta so sánh nó với phần tử ở vị trí giữa của bảng, nếu v nhỏ hơn thì nó chỉ có thể ở trong một nửa đầu tiên của bảng; nếu v lớn hơn thì nó chỉ có thể ở trong một nửa còn lại của bảng. Kế đến áp dụng đệ qui phương pháp nầy. Bởi vì chỉ gọi đệ qui một lần, chúng ta có thể đưa ra thuật toán dùng phương pháp lặp. Cài đặt dưới đây giả sử rằng mảng a đã được sắp xếp tăng theo thứ tự khóa.

```
function binarysearch(v:integer):integer;
    var x, l, r: integer;
    begin
        l := 1; r := N;
        repeat
        x := (l+r) div 2;
        if v  a[x].key
            then r := x-1
            else l := x+1;
        until (v=a[x].key) or (l>r);
        if v=a[x].key
            then binarysearch := x
            else binarysearch := N+1;
    end;
```



Hình 14.2 *Tìm kiếm nhị phân trong một tập tin lớn*

Giống như các thuật toán sắp xếp Quicksort và hoán chuyển cơ số (radix exchange), phương pháp nầy dùng các con trỏ l và r để đánh dấu tập tin con hiện đang làm việc. Mỗi khi vào vòng lặp biến x nhận giá trị điểm giữa của đoạn hiện hành, và có ba khả năng: vòng lặp kết thúc thành công, hoặc con trỏ trái được thay đổi thành x+1, hoặc con trỏ phải được đổi thành x-1, tương ứng với giá trị khóa tìm kiếm v bằng, nhỏ hơn, hay lớn hơn giá trị khóa của mẩu tin được lưu trữ ở a[x].

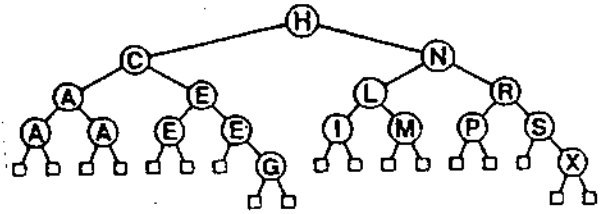
Hình 14.1 cho thấy các tập tin con được kiểm tra bởi phương pháp nầy khi tìm kiếm M trong một bảng được xây dựng từ các khóa A S E A R C H I N G E X A M P L E, khi đó kích thước đoạn giảm ít nhất một nửa ở mỗi bước, và như thế chỉ cần dùng bốn phép so sánh cho ví dụ nầy. Hình 14.2 cho thấy một ví dụ lớn hơn, với 95 mẩu tin: chỉ cần bảy phép so sánh.

TÍNH CHẤT 14.3 *Tìm kiếm nhị phân không bao giờ dùng nhiều hơn lg(N)+1 phép so sánh.*

Tính chất nầy suy ra trực tiếp từ nhận xét rằng số các mẩu tin giảm ít nhất một nửa ở mỗi bước: một cận trên của số các phép so sánh thỏa mãn hệ thức qui nạp $C_N=C_{N/2}+1$ và $C_1=1$, công thức nầy hàm ý kết quả bên trên (Công thức 2 trong chương 6).

Một điều quan trọng cần lưu ý đối với tìm kiếm nhị phân là thao tác chèn thêm một mẩu tin mới thì chiếm nhiều thời gian, bởi vì mảng phải được duy trì trong trạng thái được sắp xếp và một số mẩu tin phải được di chuyển để dành chỗ cho mẩu tin mới. Ví dụ nếu một mẩu tin mới có khóa nhỏ hơn bất cứ mẩu tin nào trong bảng thì mỗi mẩu tin phải được di chuyển lên trên một vị trí. Một thao tác chèn ngẫu nhiên đòi hỏi trung bình khoảng N/2 mẩu tin bị di chuyển. Do đó phương pháp nầy không nên dùng cho các ứng dụng đòi hỏi nhiều thao tác chèn thêm mẩu tin. Phương pháp nầy sẽ rất tốt nếu xây dựng bảng một lần (dùng Shellsort hay Quicksort) và sau đó dùng một số lớn các thao tác tìm kiếm.

Phải cẩn thận trong trường hợp các mẩu tin với khóa bằng nhau đối với thuật toán nầy: chỉ số trả về có thể rơi vào giữa một khối các mẩu tin có khóa bằng v, vì vậy phải quét theo cả hai hướng từ chỉ số đó để nhặt ra tất cả các mẩu tin có khóa bằng v. Dĩ

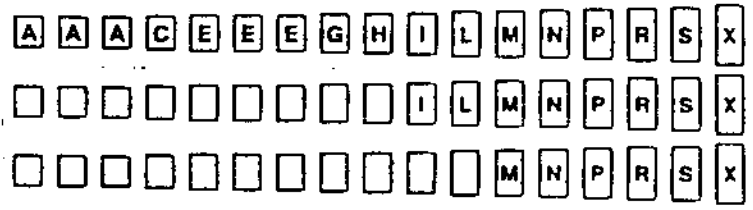


Hình 14.3 *Cây so sánh cho tìm kiếm nhị phân*

nhiên trong trường hợp nầy thời gian chạy cho việc tìm kiếm sắp xỉ với lg(N) cộng với số mẩu tin được tìm thấy.

Một dãy các phép so sánh có được bởi thuật toán tìm kiếm nhị phân được xác định trước: dãy nầy phụ thuộc vào giá trị của khóa đang được tìm kiếm và giá trị của N. Cấu trúc so sánh có thể được mô tả đơn giản bởi một cây nhị phân. Hình 14.3 cho thấy cấu trúc so sánh cho ví dụ của chúng ta. Khi tìm kiếm một mẩu tin với khóa M, trước tiên nó được so sánh với H, bởi vì M lớn hơn nên kế đến nó được so sánh với N (trường hợp ngược lại nó sẽ được so sánh với C), kế đó nó được so sánh với L, và kế đến quá trình tìm kiếm kết thúc thành công vào lần so sánh thứ tư. Sau đây chúng ta sẽ xem các thuật toán dùng cấu trúc cây nhị phân được xây dựng tường minh để điều khiển quá trình tìm kiếm.

Một cải tiến cho tìm kiếm nhị phân là cố gắng đoán chính xác hơn khóa được tìm thấy rơi vào trong đoạn nào thay vì lấy ngẫu nhiên một phần tử ở giữa. Điều nầy tương tự như một người đang tra cứu danh bạ điện thoại, nếu tên bắt đầu với chữ B, anh ta sẽ tra



Hình 14.4 *Tìm kiếm nội suy*

trong những trang đầu, nhưng nếu tên bắt đầu với Y thì anh ta sẽ tra trong những trang gần cuối. Phương pháp nầy được gọi là tìm kiếm nội suy (interpolation search), chỉ cần sửa đổi một ít chương trình bên trên. Trong chương trình trên vị trí mới để tìm kiếm (điểm giữa của đoạn) đã được tính bởi x := (l+r) div 2, mà ý nghĩa chính là biểu thức:

$$x = l + \frac{1}{2}(r-l)$$

Điểm giữa của đoạn được tính được tính bằng cách thêm một nửa kích thước đoạn vào đầu mút trái. Phương pháp nội suy thay thế con số 1/2 trong công thức trên bằng:

$$\frac{v - a[l].key}{a[r].key - a[l].key}$$

và mệnh đề x := (l+r) **div** 2 sẽ thay bởi:

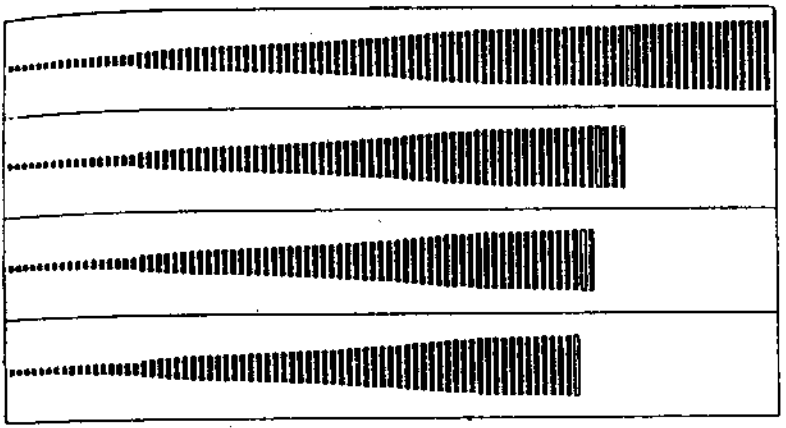
x := l + (v - a[l].key) * (r - l) **div** (a[r].key - a[l].key).

Dĩ nhiên rằng trong các biểu thức trên phải giả sử các khóa mang giá trị số.

Giả sử trong ví dụ của chúng ta chữ thứ i trong bảng chữ cái tiếng anh được đánh số là i. Khi tìm kiếm cho M, vị trí đầu tiên được kiểm tra sẽ là 9 bởi vì 1+(13-1)*(17-1)/(24-1) = 9.3 ... Việc tìm kiếm được hoàn chỉnh chỉ trong ba bước như được chỉ trong hình 14.4. Các khóa tìm kiếm khác được tìm thấy hiệu quả hơn nhiều, ví dụ phần tử đầu tiên và phần tử cuối cùng được tìm thấy ngay bước đầu tiên. Hình 14.5 cho thấy tìm kiếm nội suy trên một tập tin 95 phần tử trong hình 14.2; nó chỉ sử dụng bốn phép so sánh trong khi tìm kiếm nhị phân đòi hỏi bảy phép so sánh.

TÍNH CHẤT 14.4 *Tìm kiếm nội suy sử dụng ít hơn lglgN+1 phép so sánh trong cả hai trường hợp tìm thành công và không thành công, giả sử tìm trong các tập tin với các khóa ngẫu nhiên.*

Việc chứng minh kết quả nầy vượt ra ngoài phạm vi của quyển sách nầy. Hàm lglg(x)+1 tăng rất chậm đến nỗi có thể xem như một hằng số khi xét về mặt ứng dụng: nếu N là một tỷ thì lglgN+1<5. Do đó bất kỳ một mẩu tin đều có thể tìm thấy chỉ trong vài lần truy xuất (xét về mặt trung bình), đây là một cải tiến rất mạnh của phương pháp tìm kiếm nhị phân quen thuộc.



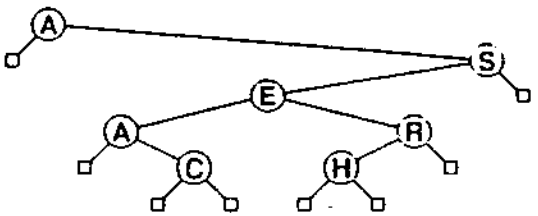
Hình 14.5 *Tìm kiếm nội suy trên một tập tin lớn*

Tuy nhiên, phương pháp tìm kiếm nội suy phụ thuộc rất nhiều vào tính phân bố tốt của các khóa trên một đoạn: nó có thể bị "lừa gạt" thê thảm khi các khóa không phân bố tốt. Mặt khác phương pháp nầy cũng đòi hỏi một vài tính toán và khi N nhỏ thì lgN đủ gần lglgN nên nó cũng không tốt hơn phương pháp tìm kiếm nhị phân nhiều lắm. Phương pháp tìm kiếm nội suy thực sự chỉ nên dùng cho các tập tin lớn, với các ứng dụng mà phép so sánh trả giá quá đắt, hay với trường hợp thao tác trên bộ nhớ ngoài.

TÌM KIẾM TRÊN CÂY NHỊ PHÂN

Tìm kiếm trên cây nhị phân là một thuật toán đơn giản, một phương pháp tìm kiếm động hiệu quả mà đủ tư cách là một trong các thuật toán nền móng trong khoa học máy tính. Thuật toán nầy sở dĩ được bàn ở đây như một thuật toán cơ bản bởi vì nó đơn giản; nhưng nó là phương pháp tìm kiếm được chọn lựa trong nhiều trường hợp ứng dụng.

Chúng ta vừa thảo luận về cây trong chương 4. Hãy ôn lại tính chất của một cây: mỗi nút chỉ được trỏ tới bởi duy nhất một nút khác gọi là cha của nó. Một cây nhị phân thì mỗi nút có hai liên kết trái và phải. Đối với việc tìm kiếm, mỗi nút của cây có một



Hình 14.6 *Một cây tìm kiếm nhị phân*

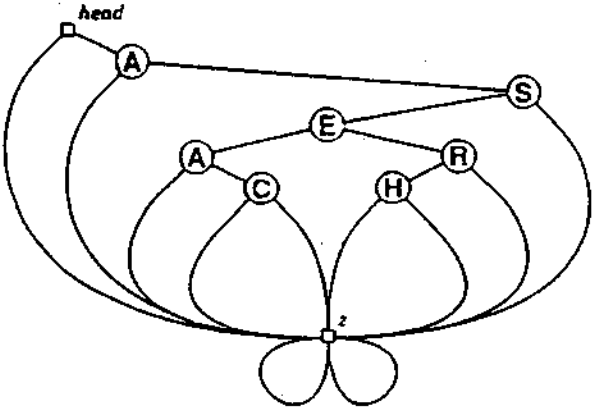
mẩu tin chứa giá trị khóa; trong một cây-tìm-kiếm-nhị-phân chúng ta giả thiết rằng tất cả các mẩu tin với các khóa nhỏ hơn thì ở trong cây-con-trái và tất cả các mẩu tin trong cây-con-phải có giá trị khóa lớn hơn hay bằng. Chúng ta thấy rằng sẽ hoàn toàn đơn giản để bảo đảm cho cây tìm kiếm nhị phân thỏa mãn định nghĩa của nó khi chèn thêm vào cây một nút mới. Một ví dụ của cây tìm kiếm nhị phân được chỉ trong Hình 14.6; như thường lệ các cây con rỗng được ký hiệu bởi các nút vuông nhỏ.

Thủ tục tìm kiếm giống như thủ tục binarysearch được hàm ý trực tiếp nhờ vào cấu trúc đã định nghĩa. Để tìm một mẩu tin với khóa v đã cho, trước tiên ta so sánh nó với nút gốc, nếu nó nhỏ hơn thì đi đến cây con trái, nếu bằng thì dừng, nếu nó lớn hơn thì đi đến cây con phải. Áp dụng đệ qui quá trình nói trên cho các cây con. Ở mỗi bước, chúng ta chắc rằng không có bộ phận nào của cây ngoài cây-con-hiện-hành có thể chứa các mẩu tin với khóa là v, và "cây con hiện hành" ngày càng nhỏ hơn. Thủ tục sẽ dừng khi có một mẩu tin với khóa v được tìm thấy hoặc "cây con hiện hành" trở nên trống nghĩa là không có một mẩu tin nào có khóa v. (Các từ "nhị phân", "tìm kiếm", và "cây" được dùng nhiều lần, độc giả nên cẩn thận phân biệt sự khác nhau giữa hàm binarysearch đã được trình bài trong phần trước đây của chương nầy và các cây tìm kiếm nhị phân được mô tả ở đây. Trong tìm kiếm nhị phân chúng ta dùng một cây nhị phân để mô tả dãy của các phép so sánh được tạo ra bởi một hàm tìm kiếm trong mảng; ở đây chúng ta xây dựng một cấu trúc dữ liệu gồm các mẩu tin được liên kết với nhau và dùng cấu trúc dữ liệu nầy cho việc tìm kiếm.)

```
type  link=↑node;
      node=record key, info: integer; l, r: link; end;
var t, head, z: link;
function treesearch(v:integr; x:link):link;
  begin
    z↑.key := v;
    repeat
      if v  x↑.key then x := x↑.l
      else x := x↑.r;
    until v=x↑.key;
    treesearch := x;
  end;
```

Ta qui ước dùng nút head mà liên kết phải của nó trỏ tới nút gốc của cây và khóa của nút head nhỏ hơn tất cả các khóa khác (để đơn giản ta cho khóa của head bằng 0 và giả sử tất cả các khóa khác là nguyên dương), liên kết trái của head thì không được dùng. Sự cần thiết của head sẽ rõ ràng hơn khi chúng ta thảo luận đến thao tác chèn.

Để tìm kiếm một mẩu tin có khóa v chúng ta cho x:=treesearch(v, head). Nếu một nút không có cây con trái (hay phải) thì liên kết trái (hay phải) của nó được trỏ tới nút "đuôi" z.



Hình 14.7 *Một cây tìm kiếm nhị phân (với các nút giả)*

Như trong trường hợp tìm kiếm tuần tự, chúng ta đặt giá trị đang muốn tìm ở trong z để dừng một quá trình tìm kiếm không thành công. Do đó, "cây con hiện hành" trỏ tới bởi x không bao giờ trở thành rỗng và mọi quá trình tìm kiếm đều là "thành công": chương trình gọi có thể kiểm tra liên kết được trả về có trỏ đến z hay không để xác định quá trình tìm kiếm có thành công hay không.

Như đã chỉ trong hình 14.6 ở bên trên, ta qui ước các liên kết trỏ tới z như là trỏ tới các nút ngoài, tất cả các quá trình tìm kiếm không thành công đều kết thúc ở các nút ngoài. Các nút thông thường có chứa các khóa được gọi là các nút trong; khi đưa thêm khái niệm nút ngoài thì lẻ ra mỗi nút trong đều phải trỏ đến hai nút khác của cây nhưng về mặt cài đặt chúng ta cho tất cả các nút ngoài được biểu diễn chỉ bởi một nút z duy nhất. Hình 14.7 cho thấy các liên kết nầy và các nút giả một cách tường minh.

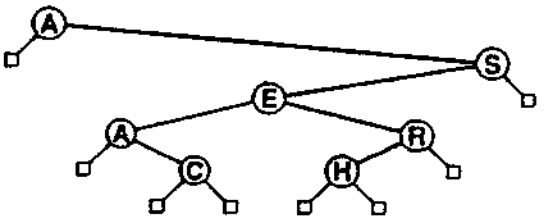
Cây rỗng được biểu diễn bởi một liên kết phải của head trỏ tới z được tạo lập bởi đoạn chương trình sau:

```
procedure treeinitialize;
  begin
  new(z); z↑.l := z; z↑.r := z;
  new(head); head↑.key := 0; head↑.r := z;
  end;
```

Khởi tạo nầy trỏ các liên kết của z đến chính nó; mặc dù các chương trình trong chương nầy không bao giờ truy xuất đến các liên kết của z, khởi tạo nầy "an toàn" và qui ước cho các chương trình nâng cao hơn mà chúng ta sẽ xét đến sau nầy.

Xét ví dụ tìm khóa I trên cây trong Hình 14.8 bằng cách dùng thủ tục treesearch. Trước tiên I được so sánh với khóa A ở gốc. Bởi vì I lớn hơn nó tiếp tục được so sánh với S, cứ tiếp tục như thế I sẽ được so sánh với E, R, và cuối cùng là H. Các liên kết trong nút chứa H trỏ tới z và quá trình tìm kiếm kết thúc: I được so sánh với chính nó ở trong z và tìm kiếm kết thúc không thành công.

Để chèn thêm một nút mới vào cây (giả sử chúng ta đã tìm kiếm nó không thành công, kế đến gặp nó trong nơi chứa z), chúng ta cần một thủ tục duy trì cha p của x trong khi duyệt cây từ gốc



Hình 14.8 *Tìm kiếm (cho khóa I) trên một cây tìm kiếm nhị phân*

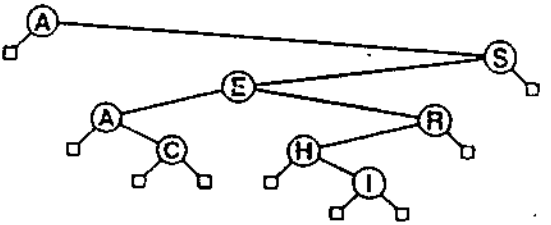
đến ngọn. Khi tiếp xúc với ngọn của cây (xảy ra x=z), p trỏ tới nút mà liên kết của nó phải được thay đổi để trỏ đến nút mới được chèn vào.

```
function treeinsert(v:integer; x:link):link;
  var p: link;
  begin
  repeat
    p := x;
    if v < x↑.key then x := x↑.l else x := x↑.r;
  until x=z;
  new(x); x↑.key := v; x↑.l := z; x↑.r := z;
  if v < p↑.key then p↑.l := x else p↑.r := x;
  treeinsert := x;
  end;
```

Một khóa v có thể được thêm vào cây bằng cách gọi hàm treeinsert(v, head). Hàm này trả về một liên kết tới nút mới được tạo sao cho thủ tục gọi có thể đặt các giá trị thích hợp vào trường info.

Khi chèn một nút mới có khóa bằng với một khóa nào đó đã có sẵn trong cây, nút mới sẽ được chèn vào bên phải của nút đã có sẵn. Tất cả các mẩu tin có khóa bằng với v có thể được xử lý bằng cách đặt liên tục t vào search(v, t) như chúng ta đã làm trong tìm kiếm tuần tự.

Cây trong Hình 14.9 có được khi chèn các khóa A S E A R C H I vào một cây trống đã được khởi động; Hình 14.10 cho thấy từng bước một khi ta thêm N G E X A M P L E vào cây trong Hình



Hình 14.9 *Chèn (khóa I) vào một cây tìm kiếm nhị phân*

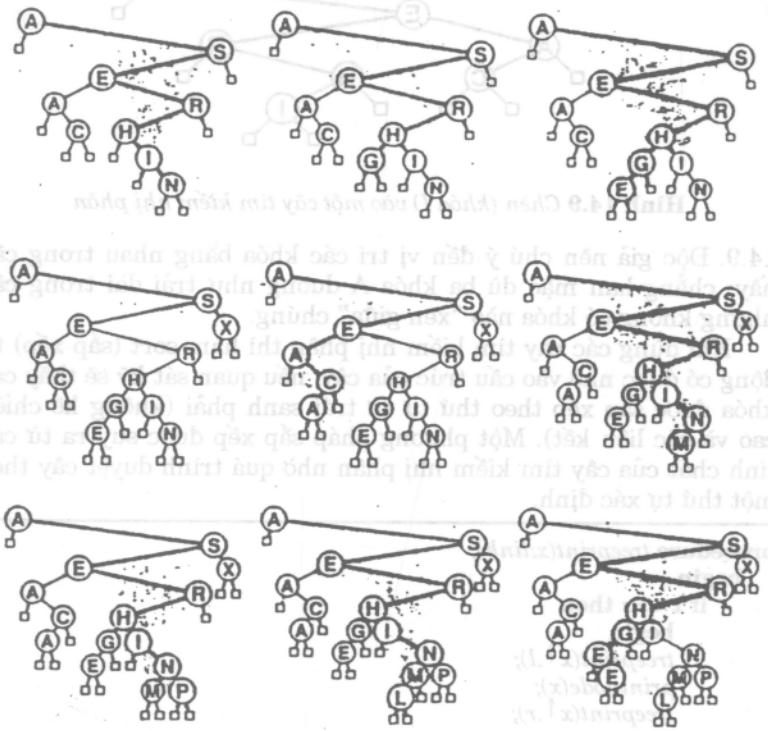
14.9. Độc giả nên chú ý đến vị trí các khóa bằng nhau trong cây nầy: chẳng hạn mặc dù ba khóa A dường như trải dài trong cây nhưng không có khóa nào "xen giữa" chúng.

Khi dùng các cây tìm kiếm nhị phân thì hàm sort (sắp xếp) tự động có được nhờ vào cấu trúc của cây, nếu quan sát kỹ sẽ thấy các khóa được sắp xếp theo thứ tự từ trái sanh phải (không kể chiều cao và các liên kết). Một phương pháp sắp xếp được suy ra từ các tính chất của cây tìm kiếm nhị phân nhờ quá trình duyệt cây theo một thứ tự xác định.

```
procedure treeprint(x:link);
  begin
    if x<>z then
      begin
      treeprint(x↑.l);
      printnode(x);
      treeprint(x↑.r);
      end;
  end;
```

Việc gọi treeprint(head↑.r) sẽ in các khóa của cây theo thứ tự. Quá trình nầy đưa ra một phương pháp sắp xếp tương tự như Quicksort, trong đó nút gốc của cây đóng vai trò phần tử phân hoạch của Quicksort. Điểm khác nhau giữa hai phương pháp sắp xếp là phương pháp sắp xếp dựa vào cây tìm kiếm nhị phân phải dùng thêm nhiều bộ nhớ trợ giúp, trong khi Quicksort dùng ít bộ nhớ trợ giúp.

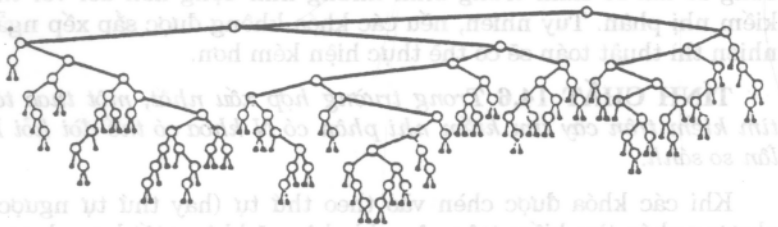
Thời gian chạy của các thuật toán trên các cây tìm kiếm nhị phân hoàn toàn phụ thuộc vào hình dạng của cây. Trong trường hợp tốt nhất, cây sẽ có hình dạng giống như Hình 14.3 với khoảng



Hình 14.10 *Xây dựng một cây tìm kiếm nhị phân*

lgN nút nằm giữa nút gốc và mỗi nút ngoài. Chúng ta hy vọng có thời gian tìm kiếm trung bình xấp xỉ logarit, bởi vì phần tử đầu tiên được chèn vào sẽ là gốc của cây; Nếu N khóa được chèn vào là ngẫu nhiên thì phần tử đầu tiên sẽ chia tập các khóa ra làm đôi (xét về mặt trung bình), lý luận tương tự cho các cây con chúng ta thấy thuật toán có thời gian tìm kiếm trung bình xấp xỉ logarit.

TÍNH CHẤT 14.5 *Một thao tác tìm kiếm hay chèn trên một cây tìm kiếm nhị phân đòi hỏi trung bình 2lnN phép so sánh,*



Hình 14.11 *Một cây tìm kiếm nhị phân lớn*

trong đó cây đang xét được xây dựng từ N khóa ngẫu nhiên.

Với mỗi nút trong cây, số các phép so sánh dùng cho thao tác tìm kiếm thành công là khoảng cách từ nút đó tới gốc. Tổng của khoảng cách này xét trên tất cả các nút được gọi là độ dài đường đi trong của cây. Chia độ dài đường đi trong cho N chúng ta có số trung bình của số lần so sánh cho trường hợp tìm kiếm thành công. Mặt khác nếu C_N ký hiệu độ dài đường đi trong của cây tìm kiếm nhị phân có N nút thì ta có công thức truy hồi:

$$C_N = N - 1 + \frac{1}{N} \sum_{1 \leq k \leq N} (C_{k-1} + C_{N-k})$$

trong đó $C_1=1$. Công thức này gần giống như công thức chúng ta đã gặp trong Chương 9 đối với Quicksort và nó có thể suy ra điều khẳng của tính chất 14.5 một cách dễ dàng. Đối với tìm kiếm không thành công thì bạn có thể làm tương tự, mặc dù nó phức tạp hơn một tí.

Hình 14.11 cho thấy một cây tìm kiếm nhị phân được xây dựng từ một hoán vị ngẫu nhiên 95 phần tử. Mặc dù nó có một số đường đi ngắn và một số đường đi dài, nó có thể được xem như hoàn toàn cân bằng: mọi thao tác tìm kiếm đều dùng ít hơn 12 phép so sánh, số lần so sánh "trung bình" để tìm thấy một khóa trong cây là 7 trong khi trường hợp tìm kiếm nhị phân là 5.74. (Số lần so sánh trung bình trong trường hợp tìm kiếm ngẫu nhiên không thành công hơn số lần trung bình trong trường hợp thành công là 1). Hơn nữa, khi một khóa mới được chèn vào thì sẽ đòi hỏi

cùng số lần so sánh trung bình nhưng linh động hơn đối với tìm kiếm nhị phân. Tuy nhiên, nếu các khóa không được sắp xếp ngẫu nhiên thì thuật toán sẽ có thể thực hiện kém hơn.

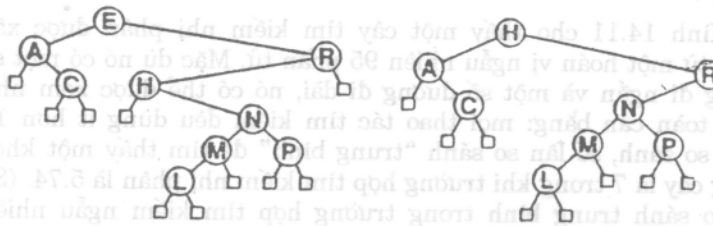
TÍNH CHẤT 14.6 *Trong trường hợp xấu nhất, một thao tác tìm kiếm trên cây tìm kiếm nhị phân có N khóa có thể đòi hỏi N lần so sánh.*

Khi các khóa được chèn vào theo thứ tự (hay thứ tự ngược), phương pháp tìm kiếm trên cây nhị phân sẽ không tốt hơn phương pháp tuần tự mà chúng ta đã thấy ở đầu chương này. Hơn nữa còn nhiều dạng cây thoái hóa khác cũng có thể đưa đến trường hợp xấu như trên (ví dụ như cây được tạo khi các khóa A Z B Y C X ... được chèn theo thứ tự đó vào một cây trống đã được khởi tạo). Trong chương tiếp theo, chúng ta sẽ xem xét một kỹ thuật để khử trường hợp xấu này và làm cho tất cả các cây gần giống như cây trong trường hợp xấu nhất.

XÓA NÚT TRÊN CÂY NHỊ PHÂN

Các cài đặt trên cho các hàm cơ bản SEARCH, INSERT, và SORT thao tác trên cây nhị phân thì hoàn toàn không có trở ngại về mặt cài đặt. Tuy nhiên cây nhị phân cũng cung cấp một ví dụ tốt đáng bàn trong các thuật toán tìm kiếm: hàm DELETE không dễ dàng cài đặt như các hàm đã xét ở trên.

Hãy xem cây bên trái trong Hình 14.12, một nút sẽ được xóa dễ dàng nếu:



Hình 14.12 *Xóa (nút E) từ một cây tìm kiếm nhị phân*

- nút đó không có con, chẳng hạn như L hay P ta tỉa nó bằng cách liên kết cha nó với null.
- nút chỉ có một con, chẳng hạn như A, H, R thì di chuyển liên kết trong con tới liên kết cha thích hợp.
- nút có hai con và một trong hai con của nó không có con, chẳng hạn như N, ta dùng nút không có con để thay cha của nó.

Nhưng trường hợp khác đối với các nút ở tầng cao hơn, chẳng hạn như nút E, thì phải xử lý như thế nào?

Hình 14.12 cho thấy một phương pháp để xóa E: thay thế nó bởi nút kế tiếp mà ở tầng cao nhất (trường hợp nầy là H). Nút nầy bảo đảm điều kiện có nhiều nhất một con (bởi vì không có nút nào giữa nó và nút bị xóa nên liên kết trái của nó phải là null), và xóa nó thì dễ dàng. Để xóa E từ cây bên trái của Hình 14.12 chúng ta trỏ liên kết trái của R tới liên kết phải của H (trường hợp nầy là N), kế đến sao chép các liên kết từ nút chứa E vào nút chứa H và trỏ head↑.r đến H. Kết quả là ta được cây bên phải trong hình vẽ.

Chương trình xét đầy đủ tất cả các trường hợp thì phức tạp hơn nhiều so với các chương trình đơn giản cho các thao tác tìm kiếm và chèn, nhưng nó rất có ích cho các thao tác trên cây phức tạp hơn mà chúng ta sẽ bàn đến trong chương sắp tới. Thủ tục sau đây xóa nút được trỏ tới bởi t từ cây có gốc là x. Do đó, một nút với khóa v có thể được xóa bằng cách gọi:

treedelete(treesearch(v, head), head).

Biến p được dùng để theo dõi cha của x ở trong cây và biến c được dùng để tìm thấy nút nối tiếp của nút bị xóa. Sau khi xóa xong, thì x là con của p. Việc giải phóng nút được trỏ tới bởi t sẽ giành cho chương trình gọi (hay có thể không giải phóng mà đưa nó vào một cấu trúc dữ liệu khác).

```
procedure treedelete(t, x:link);
   var p, c: link;
   begin
      repeat
         p := x;
         if t↑.key< x↑.key then x := x↑.l
         else x := x↑.r;
      until x=t;
      if t↑.r = z then x := x↑.l
      else  if t↑.r↑.l=z then
                  begin
                     x := x↑.r;
                     x↑.l := t↑.l;
                  end
            else
                  begin
                     c := x↑.r;
                     while c↑.l↑.l <> z do c := c↑.l;
                     x := c↑.l; c↑.l := x↑.r; x↑.l := t↑.l;
                     x↑.r := t↑.r;
                  end;
      if t↑.key < p↑.key then p↑.l := x
      else p↑.r := x;
   end;
```

Trước tiên chương trình tìm trên cây bằng phương pháp thông thường để lấy vị trí của t trên cây. Kế đến, chương trình kiểm tra ba trường hợp: nếu t không có con phải thì con của p sau khi xóa sẽ là con trái của t (đây là trường hợp như C, L, M, P, và R trong hình 14.12); nếu t có một con phải và con phải nầy không có con trái thì con phải nầy sẽ là con của p sau khi xóa và liên kết trái của nó được sao chép từ t (đây là trường hợp của A và N trong Hình 14.12); trong trường hợp ngược lại x nhận giá trị của nút có khóa nhỏ nhất trong cây con bên phải của t, liên kết phải của nút đó được sao chép vào liên kết trái của cha nó, và cả hai liên kết của nó được nhận giá trị từ t (đây là trường hợp của H và E trong Hình 14.12).

Thuật toán trên dường như không đối xứng: chẳng hạn như tại sao không sử dụng khóa đi ngay trước khóa bị xóa thay vì sử dụng một khóa đi ngay sau khóa bị xóa? Nhiều sự hiệu chỉnh đa dạng

của nó đã được đề nghị, nhưng sự khác nhau không đáng kể trong các ứng dụng thực tế. Thuật toán trên có thể dẫn đến một cây khá không cân bằng (chiều cao trung bình tương đương với sqrt(N)) nếu gặp phải một số lớn các cặp thao tác xóa-chèn ngẫu nhiên.

Trong nhiều trường hợp thì việc xóa đòi hỏi sự cài đặt phức tạp hơn, do đó người ta cũng dùng một phương pháp xóa khác gọi là "xóa lười biếng" (lazy deletion), khi một nút bị xóa thì bản thân nó vẫn còn nằm trong cấu trúc dữ liệu và bị đánh dấu xóa. Đối với phương pháp nầy lại phải chuẩn bị phương án xây dựng lại toàn bộ cấu trúc dữ liệu, bỏ đi (thực sự) những nút bị đánh dấu xóa, khi cần thiết.

CÁC CÂY TÌM KIẾM NHỊ PHÂN GIÁN TIẾP

Như chúng ta đã thấy với các heap trong Chương 11, đối với nhiều ứng dụng chúng ta muốn có một cấu trúc tìm kiếm để có thể tìm các mẩu tin một cách đơn giản mà không di chuyển chúng. Ví dụ chúng ta có một mảng a[1..N] các mẩu tin với các khóa và muốn có thủ tục tìm kiếm để lấy được chỉ số của mẩu tin có khóa bằng với một khóa đã cho trước nào đó. Đôi khi chúng ta muốn xóa mẩu tin với chỉ số đã cho từ cấu trúc tìm kiếm nhưng lại muốn giữ nó trong mảng để dùng cho mục đích khác.

Để cài đặt các cây tìm kiếm nhị phân cho các tình huống như thế, chúng ta chỉ cần cho trường info của các nút là chỉ số mảng. Kế đến chúng ta có thể loại bỏ trường key nhờ vào các chương trình tìm kiếm truy xuất các khóa trong các mẩu tin một cách gián tiếp, nghĩa là bởi một chỉ thị giống như

if v < a[x↑.info] **then** ...

Tuy nhiên, tốt hơn hết là tạo một bản sao của khóa và làm tương tự như trên. Chúng ta dùng hàm bstinsert(v, info:integer; x:link) tương tự như treeinsert, nhưng hàm bstinsert đặt giá trị đã cho của đối số vào trường info. Hàm bstdelete(v,info:integer; x:link) dùng để xóa nút với khóa v chỉ số mảng info từ cây tìm kiếm nhị phân có gốc ở x cũng bắt chước cách cài đặt của hàm treedelete. Các hàm nầy dùng một bản sao chép phụ của các khóa (một ở trong mảng, một ở trong cây), nhưng điều nầy cho phép cùng một hàm được dùng cho nhiều mảng, hoặc như sẽ thấy trong

Chương 27 nhiều trường khóa trong cùng một mảng. (Có nhiều phương pháp khác để làm được điều nầy: chẳng hạn như một thủ tục có thể được kết hợp mỗi cây được trích các khóa từ các mẩu tin.)

Một phương pháp khác để có "sự gián tiếp" cho các cây tìm kiếm nhị phân là bỏ toàn bộ các cài đặt liên kết. Nghĩa là tất cả các liên kết trở thành các chỉ số của một mảng a[1..N] các mẩu tin, mỗi mẩu tin chứa trường key và các trường chỉ số mảng l và r. Các tham chiếu liên kết chẳng hạn như

if v < x↑.key **then** x := x↑.l **else** ...

sẽ trở thành tham chiếu mảng như

if v < a[x].key **then** x := a[x].l **else** ...

Sẽ không có các lệnh gọi new trong trường hợp nầy bởi vì cây tồn tại trong mảng các mẩu tin, new(head) trở thành head := 0, new(z) trở thành z := N+1, và để chèn nút thứ M chúng ta sẽ gởi M thay vì v tới hàm treeinsert, kế đó chỉ cần tham chiếu tới a[M].key thay vì tham chiếu tới v, sau cùng thay thế dòng new(x) trong treeinsert bởi x := M.

Phương pháp cài đặt các cây tìm kiếm nhị phân nầy dùng để tìm kiếm trong các mảng lớn các mẩu tin rất ưa dùng trong nhiều ứng dụng bởi vì nó tránh sao chép các khóa như phương pháp vừa trình bày trước nó và cũng tránh dùng hàm cấp phát new nhiều lần. Khuyết điểm của phương pháp nầy là sự hao phí các khoảng bộ nhớ không dùng trong mảng các mẩu tin.

Một phương pháp thứ 3 là dùng các mảng song song như đã làm cho các xâu liên kết trong Chương 3. Sự cài đặt của phương pháp nầy rất nặng như đã mô tả trong đoạn trước, ngoài ra ba mảng được dùng, một mảng dùng cho các khóa, một dùng cho các liên kết trái và một dùng cho các liên kết phải. Thuận lợi của phương pháp nầy là tính linh động của nó. Các mảng trợ giúp (thông tin trợ giúp kết hợp với mỗi nút) có thể được thêm vào mà không cần thay đổi các thao tác trên cây, hơn nữa khi các thủ tục tìm kiếm trả về chỉ số của một nút thì ta có thể truy xuất trực tiếp đến tất cả các mảng.

BÀI TẬP

1. Cài đặt một thuật toán tìm kiếm tuần tự mà đòi hỏi trung bình khoảng N/2 bước cho cả hai trường hợp tìm kiếm thành công và không thành công, thuật toán nầy lưu các mẩu tin trong một mảng được sắp xếp.
2. Hãy cho biết thứ tự của các khóa sau khi các mẩu tin có các khóa E A S Y Q U E S T I O N được đặt vào một bảng khởi tạo rỗng bằng cách dùng hai hàm search và insert của thuật toán tìm kiếm heuristic tự quản lý.
3. Hãy đưa ra một cài đặt đệ qui của thuật toán tìm kiếm nhị phân.
4. Giả sử a[i]=2i với 1<= i <= N. Có bao nhiêu vị trí trong bảng được kiểm tra khi dùng tìm kiếm nội suy trong trường hợp tìm kiếm không thành công cho 2k - 1 ?
5. Vẽ ra cây nhị phân có được khi chèn các mẩu tin với các khóa E A S Y Q U E S T I O N vào một cây rỗng đã khởi tạo.
6. Viết một chương trình đệ qui để tính độ cao của một cây nhị phân: khoảng cách dài nhất từ gốc đến một nút ngoài.
7. Giả sử rằng chúng ta có một ước lượng trước về tần số truy xuất của các khóa trong một cây nhị phân. Các khóa nên được chèn vào theo thứ tự tăng hay giảm của tần số truy xuất? Tại sao?
8. Hãy sửa cây tìm kiếm nhị phân sao cho nó lưu tất cả các khóa bằng nhau trong cây. (Nếu bất kỳ nút khác trong cây có cùng khóa với một nút đã cho thì hoặc cha của nó hoặc một trong các con của nó nên có cùng khóa.)
9. Viết một chương trình không đệ qui để in ra các khóa trong cây tìm kiếm nhị phân theo thứ tự của khóa.
10. Vẽ một cây tìm kiếm nhị phân có được khi chèn các mẩu tin với các khóa E A S Y Q U E S T I O N vào một cây rỗng đã khởi tạo, và kế đó xóa Q.

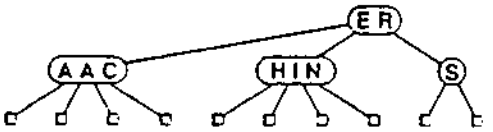
15
CÂY CÂN BẰNG

Các thuật toán về cây nhị phân trong chương trước rất tốt cho nhiều ứng dụng, tuy nhiên chúng sẽ có khuyết điểm trong trường hợp xấu nhất. Chẳng hạn như đối Quicksort, trường hợp xấu nhất của nó lại là trường hợp dễ xuất hiện trong thực tế nếu người dùng thuật toán không chú ý đến nó. Các tập tin đã được sắp thứ tự, các tập tin với thứ tự ngược, các tập tin các khóa lớn nhỏ xen lẫn nhau, hay các tập tin với sự phân đoạn lớn có cấu trúc đơn giản có thể làm thuật toán tìm trên cây nhị phân hoạt động rất tồi.

Với Quicksort, cái mà chúng ta cần để cải tiến tình huống là sắp xếp lại để có trường hợp ngẫu nhiên: bằng cách chọn một phần tử phân hoạch ngẫu nhiên chúng ta có thể dựa vào qui luật xác xuất để tránh khỏi trường hợp xấu nhất. Với tìm kiếm trên cây nhị phân thì may mắn hơn, bởi vì chúng ta có thể làm tốt hơn nhiều: có một kỹ thuật tổng quát cho phép chúng ta bảo đảm trường hợp xấu nhất sẽ không xuất hiện. Kỹ thuật nầy gọi là cân bằng, đã được dùng làm cơ sở cho nhiều thuật toán khác nhau về "cây cân bằng". Chúng ta sẽ xem xét kỹ một thuật toán thuộc loại đó và thảo luận tóm tắt về sự liên quan của nó đối với các phương pháp đã dùng khác.

Như sẽ được rõ, sự cài các thuật toán cây cân bằng là một việc "nói dễ hơn làm". Thông thường khái niệm tổng quát trước một thuật toán thì dễ dàng mô tả, nhưng cài đặt nó lại là một mớ những trường hợp đặc biệt và đối xứng. Chương trình được phát triển trong chương nầy không những là một phương pháp tìm kiếm quan trọng mà còn minh họa mối quan hệ đẹp mắt giữa một sự mô tả "cấp cao" (high-level) và một chương trình Pascal "cấp thấp" (low-level) khi cài đặt một thuật toán.

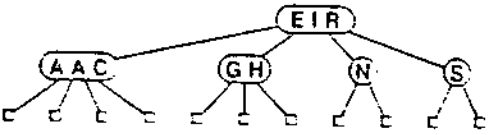
CÁC CÂY 2-3-4 TỪ TRÊN XUỐNG (Top-Down 2-3-4 Trees)



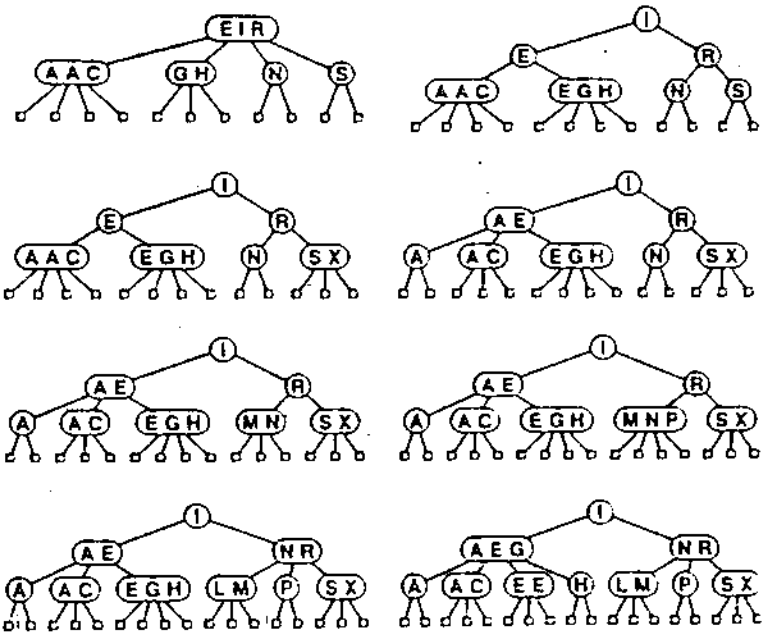
Hình 15.1 *Một 2-3-4 cây*

Để khử trường hợp xấu nhất cho cây tìm kiếm nhị phân, chúng ta cần một vài linh động trong cấu trúc dữ liệu sẽ dùng. Để có sự linh động nầy, chúng ta hãy giả sử rằng các nút trong cây của chúng ta có thể chứa nhiều hơn một khóa. Cụ thể hơn, chúng ta sẽ thừa nhận các 3-nút và 4-nút mà có thể chứa tương ứng hai và ba khóa. Một 3-nút có ba liên kết ra khỏi nó, một liên kết dành cho tất cả các mẩu tin có khóa nhỏ hơn cả hai khóa của nó, một cho tất cả các mẩu tin với các khóa nằm giữa hai khóa của nó, và một cho tất cả các mẩu tin với các khóa lớn hơn cả hai khóa của nó. Tương tự, một 4-nút có 4 liên kết đi ra khỏi nó, mỗi liên kết dành cho một đoạn trong các đoạn được định nghĩa bởi 3 khóa của nó. (Các nút trong một cây tìm kiếm nhị phân chuẩn có thể gọi là các 2-nút: một khóa và hai liên kết.) Chúng ta sẽ thấy một số phương pháp hiệu quả để định nghĩa và cài đặt các thao tác cơ sở trên các nút mở rộng nầy; bây giờ giả sử chúng ta có thể sử dụng nó một cách hình thức và xem việc tạo nên cây như thế nào.

Ví dụ trong Hình 15.1 là một 2-3-4 cây chứa các khóa A S E A R C H I N. Rất dễ dàng để thấy được cách tìm kiếm trên một cây như thế. Ví dụ, để tìm kiếm khóa G, chúng ta sẽ dò theo liên kết giữa của gốc, bởi vì G ở giữa E và R, kế đến kết thúc quá trình tìm kiếm không thành công ở liên kết trái từ nút chứa H, I, và N.

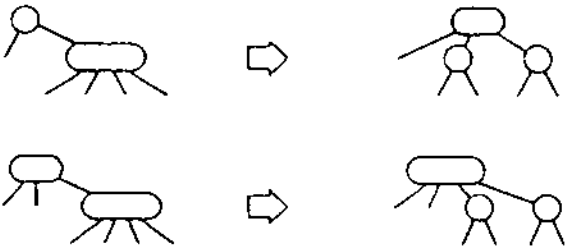


Hình 15.2 *Chèn (khóa G) vào một 2-3-4 cây*



Hình 15.3 *Xây dựng một 2-3-4 cây*

Để chèn một nút mới vào một 2-3-4 cây, chúng ta thực hiện một quá trình tìm kiếm không thành công và sau đó móc nút mới vào. Nếu kết thúc quá trình tìm kiếm ở một 2-nút thì chỉ cần sửa nó thành 3-nút. Ví dụ, X có thể được thêm vào cây trong Hình 15.1 bằng cách thêm nó (và một liên kết khác) vào nút chứa S. Tương tự một 3-nút có thể dễ dàng sửa thành 4-nút. Nhưng chúng ta sẽ làm gì để chèn một nút mới vào một 4-nút? Ví dụ làm thế nào để chèn G vào cây trong Hình 15.1? Một khả năng là treo nó vào như một con mới bên phải nhất của 4-nút chứa H, I, và N, nhưng một cách giải quyết tốt hơn được cho trong Hình 15.2: trước tiên phân rã 4-nút thành hai 2-nút và chuyển một trong các khóa của nó lên cha nó, 4-nút chứa H, I, N được tách thành hai 2-nút (một chứa H, một chứa N), "khóa giữa" I được đẩy lên 3-nút chứa E và R để đổi nó thành 4-nút. Kế đến là nơi ở của G là 2-nút chứa H.



Hình 15.4 *Tách các 4-nút*

Nhưng nếu chúng ta tách một 4-nút mà cha của nó cũng là 4-nút thì sao? Một phương pháp là cũng sẽ tách cha nó, nhưng chúng ta có thể làm mãi điều này cho đến gốc của cây. Một giải pháp dễ hơn là luôn bảo đảm cha của bất kỳ nút nào đều không là 4-nút bằng cách tách hết mọi 4-nút của cây từ trên xuống. Hình 15.3 xây dựng một 2-3-4 cây cho tập hợp các khóa A S E A R C H I N G E X A M P L E. Trên dòng đầu tiên chúng ta thấy nút gốc được tách trong suốt quá trình chèn vào của nút thứ hai E; các trường hợp tách khác xuất hiện phần tử A thứ hai, phần tử L và phần tử E thứ 3 được chèn vào.

Ví dụ trên cho thấy chúng ta có thể dễ dàng chèn các nút mới vào các 2-3-4 cây bằng cách thực hiện quá trình tìm kiếm và tách các 4-nút của cây từ trên xuống. Cụ thể như trong Hình 15.4, mỗi khi chúng ta gặp một 2-nút được nối với một 4-nút, chúng ta sẽ chuyển đổi nó thành một 3-nút được nối với hai 2-nút, và mỗi khi chúng ta chạm phải một 3-nút được nối với một 4-nút, chúng ta nên chuyển nó thành một 4-nút được nối với hai 2-nút.

Thao tác "tách" này làm việc nhiều bởi vì không chỉ các khóa mà cả các con trỏ cũng có thể bị di chuyển. Hai 2-nút có cùng số con trỏ (bốn con trỏ) như một 4-nút, nên quá trình tách có thể làm mà không thay đổi bất cứ cái gì bên dưới nút được tách. Và một 3-nút không thể được thay đổi thành 4-nút bằng cách chỉ thêm vào một khóa khác; cũng phải cần thêm một con trỏ nữa (trong trường hợp này, con trỏ trợ giúp được cung cấp bởi thao tác tách). Điều chủ yếu là các sự chuyển đổi là thuần "địa phương": không bộ

phận nào của cây cần được kiểm tra hay sửa đổi khác với cây trong Hình 14.5. Mỗi chuyển đổi chuyển một khóa từ một 4-nút lên cha của nó và xây dựng lại các liên kết tương ứng. Chú ý rằng chúng ta không cần lo ngại về nút cha là một 4-nút, bởi vì các chuyển đổi của chúng ta bảo đảm rằng khi chúng ta đi qua mỗi nút trong cây thì luôn đi ra khỏi một nút không phải là 4-nút. Cụ thể như khi chúng ta đi khỏi đáy của cây thì chúng ta không ở trên một 4-nút, và chúng ta có thể chèn nút mới trực tiếp vào cây bằng cách chuyển đổi một 2-nút thành một 3-nút hay một 3-nút thành 4-nút. Trong quá trình chèn thêm nút, chúng ta qui ước luôn tách 4-nút "ảo" ở đáy mà sắp sửa nhận thêm một nút mới. Bất cứ khi nào gốc của cây trở thành một 4-nút, chúng ta sẽ tách nó thành ba 2-nút như chúng ta đã làm cho nút đầu tiên trong ví dụ trên. Chỉ trường hợp nầy sẽ làm cây cao thêm một tầng.

Thuật toán vừa được phác thảo cung cấp một phương pháp để tìm kiếm và chèn trong 2-3-4 cây; bởi vì các 4-nút được tách từ đỉnh trở xuống, cây nói trên cũng được gọi là các cây 2-3-4 từ trên xuống. Một điều rất lý thú là mặc dù chúng ta đã không bàn đến sự cân bằng, nhưng kết quả lại là một cây hoàn toàn cân bằng!

TÍNH CHẤT 15.1 *Các thao tác tìm kiếm trong các cây 2-3-4 chứa N nút không bao giờ duyệt nhiều hơn lgN + 1 nút.*

Khoảng cách từ gốc tới mỗi nút ngoài là như nhau: các thao tác chuyển đổi mà chúng ta thực hiện không ảnh hưởng đến khoảng cách từ một nút bất kỳ tới gốc, ngoại trừ khi chúng ta tách nút gốc, trong trường hợp nầy khoảng cách từ tất cả các nút tới gốc được tăng lên một. Nếu tất cả các nút là 2-nút, kết quả đã khẳng định đúng bởi vì cây giống như cây nhị phân đầy đủ; nếu có các 3-nút và 4-nút thì chiều cao chỉ có thể ngắn hơn.

Hình 15.5 *Một cây 2-3-4 lớn*

TÍNH CHẤT 15.2 *Các thao tác chèn vào cây 2-3-4 chứa N nút cần ít hơn lgN + 1 nút trong trường hợp xấu nhất và dường như số lần tách nút trung bình thì nhỏ hơn 1.*

Trường hợp xấu nhất có thể xảy ra là tất cả các nút trên đường dẫn tới nơi chèn đều là 4-nút, tất cả chúng sẽ bị tách. Nhưng trong một cây được xây dựng từ một hoán vị ngẫu nhiên của N phần tử, không những trường hợp xấu nhất này ít xuất hiện mà ngay cả các thao tác tách về mặt trung bình cũng ít cần đến bởi vì không có nhiều 4-nút trong cây. Hình 15.5 cho thấy một cây được xây dựng từ hoán vị ngẫu nhiên của 95 phần tử: có 9 4-nút, chỉ có một nút trong chúng là không ở tầng cuối. Các kết quả phân tích về sự thực hiện trung bình của 2-3-4 cây vượt quá xa quyển sách này, nhưng đánh giá theo kinh nghiệm ta thấy rất hiếm xảy ra thao tác tách.

Sự mô tả ở trên đã đủ để định nghĩa một thuật toán dùng 2-3-4 cây để tìm kiếm, thuật toán này bảo đảm sự thực hiện tốt trường hợp xấu nhất. Tuy nhiên, chúng ta chỉ đạt được phân nửa của một cài đặt thực sự. Mặc dù có thể viết một thuật toán thực hiện các sự chuyển đổi trên các dạng dữ liệu khác nhau biểu diễn cho 2-, 3-, và 4-nút, nhưng mọi việc sẽ bị gượng ép trong biểu diễn trực tiếp này. (Có thể tin điều này bằng cách cố gắng cài đặt ngay cả trường hợp đơn giản của các sự chuyển đổi hai nút.) Hơn nữa, việc thao tác trên các cấu trúc nút quá phức tạp sẽ làm các thuật toán hoạt động chậm hơn thuật toán trên cây nhị phân thông thường. Mục đích chính của sự cân bằng là cung cấp "sự bảo hiểm" để chống lại trường hợp xấu nhất, nhưng lại vô phúc để trả giá quá đắt cho bảo hiểm đó trong mỗi lần thuật toán hoạt động. Trong phần sau, Chúng ta sẽ thấy một biểu diễn đơn giản của 2-, 3-, và 4-nút mà cho phép các chuyển đổi được làm tốt và trả giá ít hơn tìm kiếm trên cây nhị phân.

CÁC CÂY ĐỎ-ĐEN (Red-Black trees)

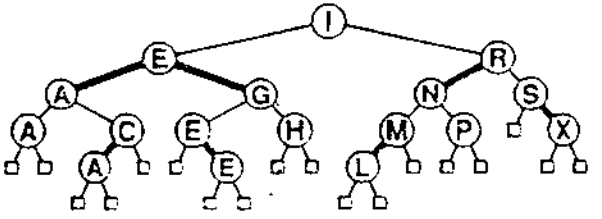
Có thể biểu diễn 2-3-4 cây như một cây nhị phân chuẩn (chỉ có các 2-nút) bằng cách dùng một bit trợ giúp cho mỗi nút. Ý tưởng là biểu diễn các 3-nút và 4-nút như các cây nhị phân nhỏ với các liên

Hình 15.6 *Biểu diễn Đỏ-Đen của các 3-nút và 4-nút*

kết "đỏ", các liên kết vốn có của 2-3-4 cây gọi là các liên kết "đen". Như trong Hình 15.6, các 4-nút được biểu diễn bởi ba 2-nút được nối bởi các liên kết đỏ và các 3-nút được biểu diễn bởi hai 2-nút được nối bởi một liên kết đỏ (các liên kết đỏ được vẽ bởi các đoạn dày nét).

Hình 15.7 cho thấy một biểu diễn của cây cuối cùng trong Hình 15.3. Nếu chúng ta bỏ đi các liên kết đỏ và kéo tất cả các nút lên ngang hàng nhau thì sẽ được cây 2-3-4 trong Hình 15.3. Bit trợ giúp cho mỗi nút được dùng để lưu trữ màu cho cho liên kết trỏ tới nút đó: chúng ta sẽ gọi đến các 2-3-4 cây biểu diễn bằng phương pháp nầy là các CÂY ĐỎ-ĐEN.

"Cách nhìn" về mỗi 3-nút được xác định bởi các thuật toán được mô tả bên dưới. Có nhiều cây đỏ đen tương ứng với mỗi 2-3-4 cây. Có thể ép buộc một qui luật là tất cả các 3-nút đều được nhìn bằng cùng một phương pháp, nhưng không có lý lẽ để làm như vậy.



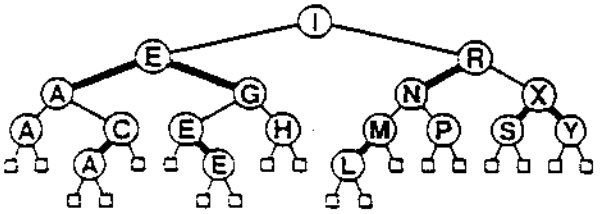
Hình 15.7 *Một cây Đỏ-Đen*

Những cây nầy có các tính chất cấu trúc được suy ra từ cách định nghĩa chúng. Ví dụ, không bao giờ có hai liên kết đỏ kề nhau dọc theo bất kỳ con đường nào từ gốc tới một nút ngoài, và tất cả các con đường đều có số liên kết đen bằng nhau. Chú ý rằng có thể một con đường (luân phiên đen-đỏ) dài gấp hai lần một con đường khác (tất cả đen), nhưng tất cả các độ dài đường đi đều xấp xỉ logN.

Một chức năng nổi bật của cây nói trên là vị trí của các khóa bằng nhau. Thuật toán trên cây cân bằng sẽ làm cho các mẩu tin có khóa bằng với một nút cho trước rơi vào cả hai phía của nút đó: ngược lại, trường hợp không cân bằng có thể gây ra một chuỗi dài các bằng nhau. Điều nầy hàm ý rằng chúng ta không thể tìm thấy tất cả các nút với một khóa đã cho bằng cách gọi lập lại các thủ tục tìm kiếm như trong chương trước. Tuy nhiên, điều nầy không quan trọng bởi vì đối với tất cả các nút trong cây con mà có cùng khóa với nút gốc của cây con thì đều có thể được tìm thấy bởi một thủ tục đệ qui đơn giản giống như thủ tục treeprint của chương trước.

Một tính chất rất đẹp của các cây đỏ-đen là thủ tục treesearch để tìm kiếm trên cây nhị phân chuẩn sẽ không cần sửa đổi (ngoại trừ đối với các khóa bằng nhau vừa được thảo luận trong đoạn trước). Chúng ta sẽ cài đặt các màu liên kết bằng cách thêm một trường luận lý (boolean) tên là red vào mỗi nút, trường nầy sẽ là true nếu liên kết trỏ tới nút là đỏ và là false nếu liên kết là đen; thủ tục treesearch không bao giờ truy xuất tới trường red. Trong một ứng dụng, thông thường mỗi khóa được chèn vào chỉ một lần nhưng có thể được tìm kiếm nhiều lần, do đó chúng ta cải tiến được thời gian tìm kiếm bởi vì cây cân bằng và không cần thêm các thao tác để duy trì sự cân bằng của cây trong suốt các quá trình tìm kiếm.

Hơn nữa khả năng tách rất hiếm trong quá trình chèn, chúng ta chỉ làm thêm một số việc khi thấy các 4-nút, và trong cây không có nhiều 4-nút bởi vì chúng ta luôn tách chúng ra. Trong cài đặt sau đây của thủ tục chèn, ta thấy chỉ cần một kiểm tra trợ giúp trong vòng lặp (nếu một nút có hai con đỏ, nó là một bộ phận của một 4-nút).

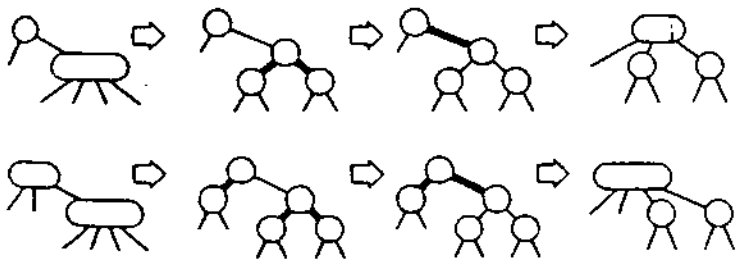


Hình 15.8 *Chèn Y vào một cây Đỏ-Đen*

```
function rbtreeinsert(v: integer; x: link): link;
  var gg, g, p: link;
  begin
  p:=x; g:=x;
  repeat
    gg:=g; g:=p; p:=x;
    if x < x↑.key then x:=x↑.l else x:=x↑.r;
    if x↑.l↑.red and x↑.r↑.red then x:=split(v, gg, g, p, x);
  until x=z;
  new(x); x↑.key:=v; x↑.l:=z; x↑.r:=z;
  if v < p↑.key then p↑.l:=x else p↑.r:=x;
  rbtreeinsert:=x;
  x:=split(v, gg, g, p, x);
  end;
```

Trong chương trình này, x di chuyển hướng xuống dưới cây như trước đây và gg, g, và p được lưu để trỏ đến ông cố, ông nội và cha của x ở trong cây. Để có thể thấy tại sao phải cần các liên kết này, hãy xem việc thêm Y vào cây ở trên. Khi gặp nút ngoài bên phải của 3-nút chứa S và X, gg là R, g là S, và p là X. Bây giờ Y phải được thêm vào để tạo một 4-nút chứa S, X, và Y, kết quả được cho thấy như trong Hình 15.8.

Chúng ta cần một con trỏ tới R (gg) bởi vì liên kết phải của R phải được thay đổi để trỏ tới X, chứ không phải S. Để thấy chính xác điều này, chúng ta cần quan sát thao tác của thủ tục split. Hãy xem biểu diễn đỏ-đen cho hai phép biến đổi mà chúng ta phải thực hiện: nếu chúng ta một 2-nút nối tới một 4-nút, thì chúng ta sẽ đổi chúng thành một 3-nút nối tới hai 2-nút; nếu chúng ta có một 3-nút nối tới một 4-nút, thì chúng ta nên đổi chúng thành một

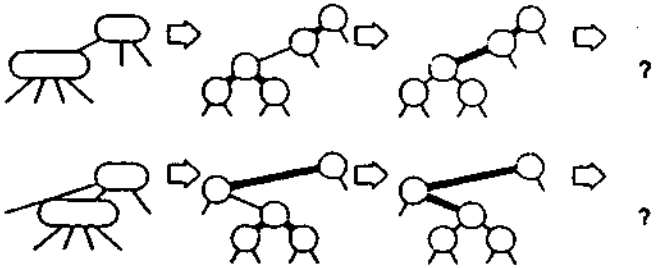


Hình 15.9 *Tách các 4-nút*

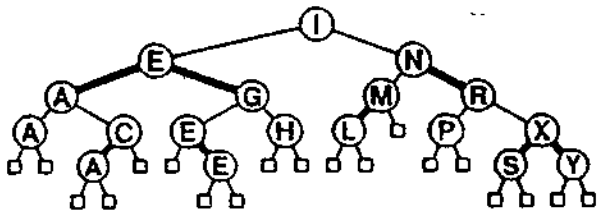
4-nút nối với hai 2-nút. Khi một nút mới được thêm vào đáy, nó được xem là nút giữa của một 4-nút ảo (nghĩa là suy nghĩ về z như red mặc dù điều này không được kiểm tra một cách tường minh).

Sẽ dễ dàng chuyển đổi nếu chúng ta gặp một 2-nút nối với một 4-nút, và cũng như vậy nếu chúng ta có một 3-nút nối với một 4-nút theo cách nối "bên phải" như trong Hình 15.9. Do đó, thủ tục split bắt đầu bằng cách đánh dấu x là đỏ và con của x là đen.

Nếu chúng ta gặp một 3-nút nối tới một 4-nút như trong Hình 15.10 thì sẽ có hai trường hợp xảy ra (Thật ra có bốn trường hợp xảy ra, bởi vì các ảnh đối xứng của hai trường hợp này cũng có thể xảy ra đối với các 3-nút có phương khác). Trong những trường hợp này, việc tách các 4-nút giữ lại hai nút đỏ trên cùng một hàng, đây là một trường hợp không hợp lệ cần phải được giải quyết. Trường



Hình 15.10 *Tách các 4-nút, cần đến phép quay*



Hình 15.11 *Quay một 3-nút trong Hình 15.8*

hợp này được kiểm tra dễ dàng trong chương trình: chúng ta phải đánh dấu x là đỏ, vì vậy nếu cha p của x cũng đỏ thì chúng ta phải làm thêm thao tác. Tình huống này không tồi bởi vì chúng ta có ba nút được nối bằng các liên kết đỏ, tất cả những gì mà chúng ta cần làm là chuyển cây sao cho các liên kết đỏ chỉ xuống dưới từ cùng một nút.

Thật may mắn, có một thao tác đơn giản để có được điều mong muốn. Chúng ta hãy bắt đầu với trường hợp dễ hơn trong hai trường hợp nói trên, trường hợp đầu tiên trong Hình 15.10, trong đó các liên kết đỏ có cùng hướng. Vấn đề là 3-nút được hướng theo phương ngược lại, nên chúng ta xây dựng lại cây để đổi hướng của 3-nút, và do đó đưa trường hợp này về trường hợp thứ hai trong Hình 15.9 chỉ cần đổi màu của x và các con của nó là xong. Việc xây dựng lại cây để đổi hướng một 3-nút bao gồm thay đổi ba liên kết như trong Hình 15.11; chú ý rằng Hình 15.11 giống như Hình 15.8 nhưng nó có 3-nút chứa N và R được quay. Liên kết trái của R được thay đổi để trỏ tới P, liên kết phải của N được thay đổi để trỏ tới R, và liên kết phải của I được thay đổi để trỏ tới N. Cũng vậy, cần thận chú ý rằng màu của hai nút cũng bị đổi.

Thao tác quay đơn (single rotation) được định nghĩa trên cây tìm kiếm nhị phân bất kỳ (nếu chúng ta không chú ý các thao tác về màu) và là cơ sở cho nhiều thuật toán về cây cân bằng, bởi vì nó giữ gìn đặc trưng cần thiết của cây tìm kiếm và là sự sửa đổi địa phương bao gồm chỉ ba thao tác thay đổi liên kết. Tuy nhiên một chú ý rất quan trọng là một thao tác quay đơn không nhất thiết cải tiến sự cân bằng của cây. Trong Hình 15.11, thao tác quay mang

tất cả các nút bên trái của N gần gốc hơn một tầng so với trước đó, nhưng tất cả các nút bên phải của R thì bị thấp hơn một tầng, trường hợp nầy làm cây ít cân bằng hơn. Các cây 2-3-4 từ trên xuống có thể được xem đơn giản như một phương tiện qui ước để đồng nhất các thao tác quay đơn để cải tiến sự cân bằng.

Việc làm một thao tác quay đơn bao gồm sự sửa đổi cấu trúc của cây, và điều nầy phải được làm cẩn thận. Như đã thấy khi xem thuật toán xóa trong Chương 14, chương trình thì dường như phức tạp hơn cần thiết bởi vì có một số trường hợp tương tự với các đối xứng trái-phải. Chẳng hạn như giả sử rằng các liên kết y, c, và gc trỏ tương ứng tới I, R, và N trong Hình 15.8. Khi đó sự chuyển đổi tới Hình 15.11 được gây ra bằng các thay đổi liên kết:

c↑.l:=gc↑.r; gc↑.r := c; y↑.r := g;

Có ba trường hợp tương tự khác: 3-nút có thể được định hướng theo cách khác, hay nó có thể là bên trái của y. Một phương pháp truyền thống để xử lý bốn trường khác nhau là dùng việc tìm kiếm khóa v để "tìm trở lại" con trực tiếp c và cháu nội gc của nút y (chúng ta đã biết rằng chỉ định lại hướng một 3-nút nếu tìm đến nút đáy). Nhờ điều đó mà chúng ta có chương trình đơn giản hơn phải nhớ không những hai liên kết tương ứng với c và gc mà còn phải nhớ chúng nó là các liên kết phải hay liên kết trái trong suốt quá trình tìm kiếm. Chúng ta có hàm sau đây để định hướng lại một 3-nút dựa theo con đường tìm kiếm cho v mà cha của nó là y:

```
function rotate(v:integer; y:link):link;
  var c, gc: link;
  begin
  if v<y↑.key then c := y↑.l else c := y↑.r;
  if v<c↑.key
     then begin gc := c↑.l; c↑.l := gc↑.r; gc↑.r := c; end
     else begin gc := c↑.r; c↑.r := gc↑.l; gc↑.l := c; end;
  if v<y↑.key then y↑.l := gc else y↑.r := gc;
  rotate := gc;
  end;
```

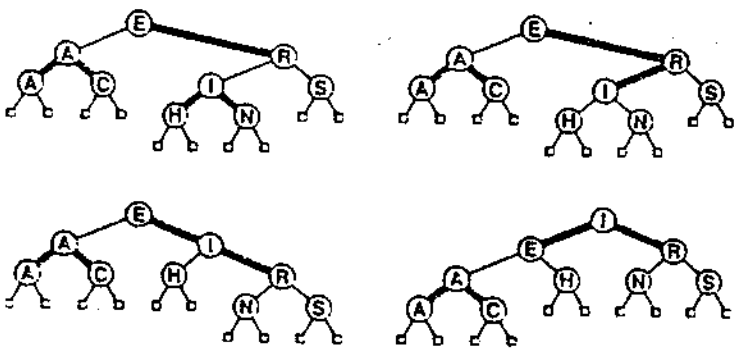
Nếu y trỏ tới gốc, c là liên kết phải của y và gc là liên kết trái của c, điều nầy thực hiện chính xác các chuyển đổi liên kết cần để sản sinh ra cây trong Hình 15.11 từ Hình 15.8. Độc giả có lẻ muốn kiểm tra các trường hợp khác. Hàm nầy trả về liên kết tới đỉnh của 3-nút, nhưng lại không làm chính thao tác đổi màu.

Do đó, để xử lý trường hợp thứ ba cho thủ tục split (xem Hình 15.10), chúng ta có thể làm đỏ g, cho x nhận giá trị rotate(v.gg), và làm đen x. Thao tác này đổi hướng của 3-nút bao gồm hai nút được trỏ tới bởi g và p, đồng thời nó cũng đưa trường hợp này trở về giống như trường hợp thứ hai khi 3-nút hướng về bên phải.

Cuối cùng khi hai liên kết đỏ hướng tới hai phương khác nhau (xem Hình 15.10), chúng ta chỉ cần cho p nhận giá trị rotate(v.g). Thao tác này định lại hướng của 3-nút "không hợp lệ" bao gồm bao gồm hai nút được trỏ tới bởi p và x. Các nút này có cùng màu, vì thế không cần thiết phải đổi màu chúng, và chúng ta trở về trường hợp thứ ba. Phối hợp thao tác này và phép quay của trường hợp thứ ba được gọi là **sự quay kép (double rotation)**.

Hình 15.12 cho thấy thao tác split xuất hiện trong ví dụ của chúng ta khi G được thêm vào. Trước tiên, thao tác đổi màu để tách 4-nút chứa H, I, N. Kế đến là một thao tác quay kép: bộ phận đầu tiên xung quanh cạnh giữa I và R, bộ phận thứ hai xung quanh cạnh giữa E và I. Sau những sửa đổi này, G có thể được chèn vào bên trái của H như trong cây đầu tiên trong Hình 15.13.

Hàm sau đây sẽ bổ sung đủ các thao tác được thực hiện bởi split. Nó phải chuyển các màu của x và các con của nó, làm phần dưới của một thao tác quay nếu cần thiết và kế đến thực hiện sự quay đơn nếu cần thiết.



Hình 15.12 *Tách một nút trong một cây Đỏ-Đen*

```
function split(v:integer; gg,g,p,x:link):link;
   begin
   x↑.red := true; x↑.l↑.red := false; x↑.r↑.red := false;
   if p↑.red then
   begin
      g.red := true;
      if (v < g↑.key) <> (v < p↑.key) then p := rotate(v,g);
      x := rotate(v,gg);
      x↑.red := false;
      end;
   head↑.r↑.red := false;
   split := x;
   end;
```

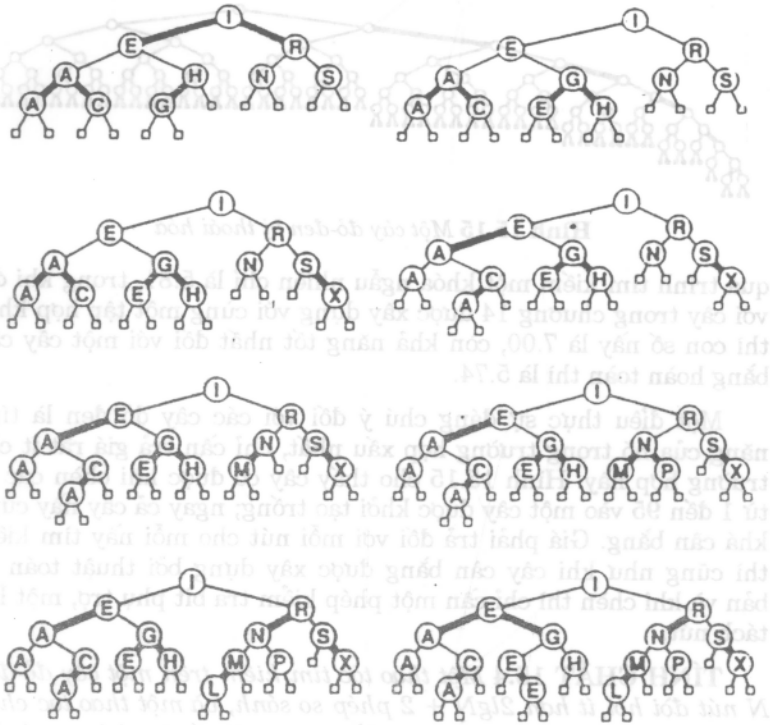
Nếu gốc là một 4-nút thì thủ tục split làm cho gốc có màu đỏ: thao tác nầy tương ứng với sự chuyển đổi nó cùng với nút nháp (dummy) ở phía trên nó thành một 3-nút. Đương nhiên không có lý do gì để làm điều nầy, vì vậy có một lệnh ở cuối của thủ tục split dùng để duy trì màu đen của gốc. Khi bắt đầu xử lý, cần phải khởi tạo cẩn thận nút nháp như đoạn chương trình sau đây:

```
type link = ↑node;
   node = record key,info: integer; l,r: link; red: boolean; end;
var head,z: link;
procedure rbtreeinitialize;
   begin
   new(z); z↑.l := z; z↑.r := z; z↑.red := false;
   new(head); head↑.key := 0; head↑.r := z;
   end;
```

Hình 15.13 cho thấy cây đỏ-đen được tạo từ tập hợp khóa mẫu nhờ vào thuật toán nầy. Ở đây, chỉ trả giá một vài phép quay mà chúng ta có được một cây cân bằng hơn rất nhiều so với cây trong chương 14 mà được tạo từ cùng một tập hợp khóa.

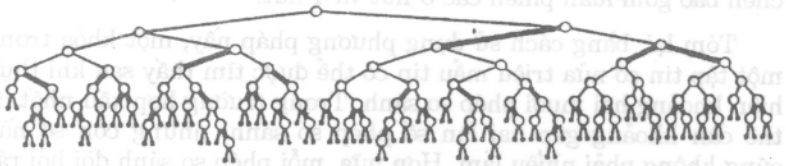
TÍNH CHẤT 15.3 *Một thao tác tìm kiếm trên cây đỏ-đen với N nút được xây dựng từ các khóa ngẫu nhiên có lẻ đòi hỏi khoảng lgN phép so sánh, và về mặt trung bình thì một thao tác có lẻ chèn đòi hỏi ít hơn một phép quay.*

Sự phân tích chính xác trường hợp trung bình của thuật toán nầy thì chưa được thực hiện, nhưng có các kết quả thuyết phục từ

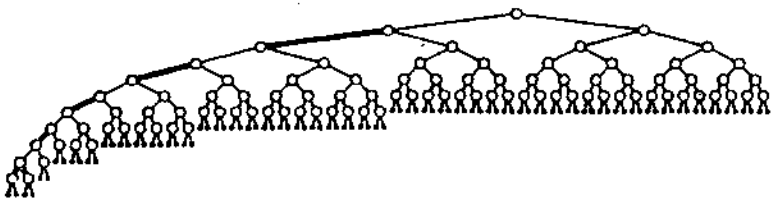


Hình 15.13 *Xây dựng một cây đỏ-đen*

các phân tích các trường hợp riêng và các trường hợp giả lập. Hình 15.14 cho thấy một cây được tạo từ ví dụ lớn hơn ví dụ của chúng ta đang dùng: số lượng nút trung bình được duyệt qua trong suốt



Hình 15.14 *Một cây đỏ-đen lớn*



Hình 15.15 *Một cây đỏ-đen bị thoái hóa*

quá trình tìm kiếm một khóa ngẫu nhiên chỉ là 5.81, trong khi đối với cây trong chương 14 được xây dựng với cùng một tập hợp khóa thì con số nầy là 7.00, còn khả năng tốt nhất đối với một cây cân bằng hoàn toàn thì là 5.74.

Một điều thực sự đáng chú ý đối với các cây đỏ đen là tính năng của nó trong trường hợp xấu nhất, chỉ cần trả giá rất ít cho trường hợp nầy. Hình 15.15 cho thấy cây có được khi chèn các số từ 1 đến 95 vào một cây được khởi tạo trống; ngay cả cây nầy cũng khá cân bằng. Giá phải trả đối với mỗi nút cho mỗi nầy tìm kiếm thì cũng như khi cây cân bằng được xây dựng bởi thuật toán cơ bản và khi chèn thì chỉ cần một phép kiểm tra bit phụ trợ, một lần tách nút.

TÍNH CHẤT 15.4 *Một thao tác tìm kiếm trên một cây đỏ-đen N nút đòi hỏi ít hơn $2\lg N + 2$ phép so sánh, và một thao tác chèn đòi hỏi số phép quay ít hơn một phần tư so với số các phép so sánh.*

Chỉ có các thao tác "tách" một 3-nút được nối với một 4-nút trong một cây 2-3-4 đòi hỏi một thao tác quay tương ứng như cây đỏ đen, vì vậy tính chất nầy được suy ra từ tính chất 15.2. Trường hợp xấu nhất xảy ra khi đường dẫn tới điểm thực hiện thao tác chèn bao gồm luân phiên các 3-nút và 4-nút.

Tóm lại, bằng cách sử dụng phương pháp nầy, một khóa trong một tập tin cỡ nửa triệu mẩu tin có thể được tìm thấy sau khi thực hiện khoảng hai mươi phép so sánh. Trong trường hợp xấu nhất có thể cần khoảng gấp hai lần số phép so sánh, nhưng con số nầy cũng không phải nhiều lắm. Hơn nữa, mỗi phép so sánh đòi hỏi rất ít thời gian vì vậy thuật toán tìm kiếm sẽ rất nhanh.

CÁC THUẬT TOÁN KHÁC

Sự cài đặt các thuật toán cho "cây 2-3-4 từ trên xuống" bằng cách dùng khuôn mẫu của cây đỏ-đen cho trong phần trước là một trong nhiều chiến lược tương tự như đã làm trong việc cài đặt cho các cây nhị phân cân bằng. Như chúng ta đã thấy ở trên, thật ra các thao quay đã làm cân bằng cây: chúng ta đã nhìn cây với một khía cạnh đặc biệt làm cho nó thích hợp với thao tác quay. Khi chú ý đến khía cạnh khác của cây thì sẽ dẫn đến các thuật toán khác, chúng ta sẽ mô tả tóm tắt một số thuật toán như thế.

Cấu trúc dữ liệu nổi tiếng nhất và cổ điển nhất đối với những cây cân bằng là các cây AVL. Những cây nầy có tính chất là chiều cao hai cây con của mỗi nút thì sai khác nhau nhiều nhất là 1. Nếu điều kiện nầy bị vi phạm do một thao tác chèn nào đó thì cây phải được cập nhật lại bằng cách dùng các thao tác quay. Điều nầy đòi hỏi thêm vào một vòng lặp: dùng thuật toán cơ sở để tìm vị trí cho giá trị muốn chèn vào, kế đó tiếp tục duyệt hướng lên phía trên cây dọc theo con đường đã duyệt trước đó, và hiệu chỉnh độ cao của các nút bằng cách ngừng các thao tác quay. Cũng cần biết xem mỗi nút có chiều cao nhỏ hơn một, bằng, hay lớn hơn một so với nút anh em của nó. Điều nầy đòi hỏi phải mã hóa bằng 2 bit nếu bằng phương pháp thông thường, nhưng nếu dùng một khuôn mẫu giống như các cây đỏ-đen thì lại chỉ cần 1 bit.

Một cấu trúc cây cân bằng nổi tiếng thứ hai là cây 2-3, cây nầy chỉ thừa nhận các 2-nút và 3-nút. Có thể cài đặt thủ tục insert bằng dùng một "vòng lặp thêm vào" bao gồm các thao tác quay như đối với các cây AVL, nhưng lại không thể theo kiểu trên-xuống như trước đây. Một lần nữa, khuôn mẫu giống như các cây đỏ-đen được dùng để cài đặt, nhưng sẽ thật sự tốt hơn khi dùng các cây 2-3-4 từ dưới lên, trong trường hợp nầy chúng ta tìm kiếm tới đáy của cây và chèn vào đó, kế đến (nếu nút đáy là một 4-nút) di chuyển trở lên phía trên con đường vừa duyệt, tách các 4-nút và chèn nút giữa vào nút cha, đến khi chạm một 2-nút hay 3-nút mà là nút cha thì có thể thực hiện một phép quay để điều chỉnh trường hợp giống như trong hình 15.10. Phương pháp nầy có thuận lợi là dùng nhiều nhất một phép quay đối với mỗi thao tác chèn và đây có thể thuận tiện đối với một vài ứng dụng. Sự cài đặt

thì phức tạp hơn một ít so với phương pháp trên xuống đã bàn đến ở trên.

Trong chương 18, chúng ta sẽ nghiên cứu dạng quan trọng nhất đối với cây cân bằng, một ở rộng của các cây 2-3-4 mà được gọi là các B-cây. Những cây nầy cho phép chứa M khóa mỗi nút đối với M lớn và được dùng rộng rãi cho các ứng dụng tìm kiếm bao gồm các tập tin rất lớn.

BÀI TẬP

1. Hãy vẽ một cây 2-3-4 từ trên xuống có được khi chèn các khóa E A S Y Q U E S T I O N theo thứ tự đó vào một cây được khởi tạo trống.

2. Hãy vẽ một biểu diễn đỏ-đen của cây trong câu hỏi trước.

3. Chính xác các liên kết nào được sửa đổi bởi các thủ tục split và rotate khi Z được chèn vào (sau Y) vào cây ví dụ cho chương này.

4. Vẽ cây đỏ-đen có được khi chèn các ký tự từ A đến K theo thứ tự, và mô tả nói chung điều gì xảy ra khi các khóa được chèn vào theo thứ tự tăng.

5. Thông thường có bao nhiêu liên kết của cây phải bị thay đổi đối với một thao tác quay kép, và bao nhiêu bị thay đổi trong một cài đặt đã cho.

6. Hãy phát sinh hai cây đỏ-đen ngẫu nhiên 32-nút, vẽ chúng (bằng tay hay bằng chương trình), và so sánh chúng với các cây tìm kiếm nhị phân không cân bằng được xây dựng với cùng một tập hợp khóa đó.

7. Hãy phát sinh 10 cây đỏ-đen ngẫu nhiên 1000-nút. Tính số phép quay cần thiết để xây dựng các cây và khoảng cách trung bình từ nút gốc tới một nút ngoài cho mỗi cây. Hãy thảo luận các kết quả.

8. Với một bit màu cho mỗi nút, chúng ta có thể biểu diễn 2-nút, 3-nút, và 4-nút. Bao nhiêu dạng nút khác nhau có thể có thể có nếu chúng ta dùng hai bit cho mỗi nút.

9. Các thao tác quay được đòi hỏi trong các cây đỏ-đen khi các 3-nút bị đổi thành các 4-nút bởi một phương pháp "không cân bằng". Tại sao ta không khử bỏ các phép quay bằng cách cho phép các 4-nút được biểu diễn như ba nút bất kỳ được nối bởi hai liên kết đỏ (hoàn toàn cân bằng hay không)?

10. Hãy cho một dãy thao tác chèn mà sẽ xây dựng cây đỏ-đen được chỉ trong hình 15.11.

16
PHÉP BĂM (HASHING)

Một cách tiếp cận đầy đủ tới việc tìm kiếm khác với tìm kiếm trên các cấu trúc cây dựa so sánh của chương trước là phép băm: một phương pháp tham chiếu trực tiếp đen các mẩu tin bằng cách thực hiện các phép chuyển đổi số học từ các khóa vào các địa chỉ bảng. Nếu chúng ta biết rằng các khóa là các số nguyên phân biệt từ 1 đến N thì chúng ta có thể lưu mẩu tin với khóa i trong vị trí i của bảng, chuẩn bị để truy xuất tức thời nhờ vào giá trị khóa. Phép băm là sự tổng quát hóa của phương pháp tầm thường nầy cho các ưng dụng tìm kiếm thông thường khi chúng ta không có sự hiểu biết đặc biệt về khóa (chưa có thông tin cụ thể về các giá trị khóa).

Bước đầu trên của việc tìm kiếm bằng phép băm là tính một hàm băm (hash function) để chuyển đổi từ khóa tìm kiếm vào địa chỉ bảng. Trường hợp lý tưởng, các khóa khác nhau nên ánh xạ vào các địa chỉ khác nhau, nhưng thực tế thì không có hàm băm hoàn chỉnh và sẽ có hai hay nhiều khóa khác nhau sẽ băm đến cùng một địa chỉ. Phần thứ hai của tìm kiếm băm là giải quyết xung đột (collision-resolution) để cư xử với các khóa như đã nói. Một trong các phương pháp giải-quyết-xung-đột mà chúng ta nghiên cứu là dùng các danh sách liên kết, bởi vì lưu trữ động nên phương pháp nầy thích hợp khi số lượng khóa tìm kiếm không thể tiên đoán trước. Hai phương pháp xử lý xung đột khác mà chúng ta xem xét sẽ có thời gian tìm kiếm nhanh trên các mẩu tin được lưu trữ trong một mảng cố định.

Phép băm là một thí dụ tốt về vấn đề dung hòa giữa thời gian chạy và dung lượng bộ nhớ sử dụng. Nếu không có sự giới hạn về bộ nhớ thì chúng ta có thể thực hiện bất kỳ một thao tác tìm kiếm nào chỉ với một lần truy xuất bộ nhớ bằng cách sử dụng khóa như một địa chỉ bộ nhớ. Nếu không có sự giới hạn về thời gian thì chúng ta có thể tối thiểu hóa dung lượng bộ nhớ sử dụng bằng cách

dùng một phương pháp tìm kiếm tuần tự. Phép băm cung cấp một phương pháp dùng một lượng vừa phải của bộ nhớ và để làm một sự cân bằng giữa hai thái cực này. Sử dụng hiệu quả bộ nhớ có sẵn và truy xuất nhanh đến bộ nhớ là quan tâm chủ yếu của bất kỳ một phương pháp băm.

Phép băm là một bài toán "cổ điển" của khoa học máy tính đã có nhiều thuật toán khác nhau được nghiên cứu và được dùng rất rộng rãi. Có một số lượng lớn những phân tích và kinh nghiệm để cung cấp các thủ tục băm cho rất nhiều ứng dụng khác nhau.

CÁC HÀM BĂM

Vấn đề đầu tiên là chúng ta phải tính toán hàm băm để chuyển đổi các khóa thành các địa chỉ bảng. Đây là một tính toán số học có các tính chất tương tự như các bộ phát sinh số ngẫu nhiên mà chúng ta sẽ nghiên cứu trong chương 33. Chúng ta cần một hàm chuyển đổi các khóa (các khóa có thể là những số nguyên hay các ký tự ngắn) thành các số nguyên trong khoảng [0..M-1] trong đó M là số các mẩu tin mà có thể chứa đủ trong số lượng bộ nhớ có sẵn. Một hàm băm lý tưởng là một hàm mà dễ dàng tính và gần giống như một hàm "ngẫu nhiên".

Bởi vì các phương pháp mà chúng ta sẽ dùng là các phương pháp số học, bước đầu tiên là chuyển các khóa thành các số để có thể thực hiện các phép toán số học. Với các khóa nhỏ thì hầu như không cần làm gì cả bởi vì chúng ta sẽ dùng biểu diễn nhị phân của các khóa như những con số (xem thảo luận ở đầu chương 10). Với các khóa lớn hơn, người ta phải suy nghĩ cách xóa các bit khỏi các chuỗi ký tự và nén tất cả chúng lại thành một từ cơ sở của máy tính; tuy nhiên chúng sẽ thấy một phương pháp chung cho các khóa có chiều dài tùy ý.

Trước tiên giả sử rằng chúng ta có một số nguyên lớn tương ứng trực tiếp tới khóa của chúng ta. Có lẻ phương pháp được dùng duy nhất cho phép băm là chọn M nguyên tố và với mọi k ta tính h(k)=k mod M. Đây là một phương pháp đơn giản rất dễ dàng tính trong nhiều môi trường và trải khá tốt vào tập các giá trị khóa.

Ví dụ kích thước bảng của ta là 101 và cần phải tính một chỉ số cho một khóa bốn ký tự A K E Y: nếu khóa được mã hóa bằng mã 5 bit đơn giản như trong chương 10 (ký tự thứ i trong bảng chữ cái được biểu diễn bởi biểu diễn nhị phân của số i) thì chúng ta có thể xem khóa trên như số nhị phân 00001010110010111001, số này tương đương với 44217 thập phân. Bây giờ ta có 44217 thập phân. Bây giờ ta có 44217 = 80(mod 101), vì vậy khóa A K E Y được áp tới vị trí 80 trong bảng. Vì có rất nhiều giá trị khóa nhưng lại ít các vị trí trong bảng nên có khả năng các khác nhau được áp tới cùng một vị trí trong bảng (ví dụ như khóa B A R H cũng có địa chỉ 80 theo cách băm nói trên).

Tại sao kích thước M của bảng băm phải nguyên tố ? Trả lời cho câu hỏi này phụ thuộc vào tính chất số học của hàm mod. Chúng ta đang dùng khóa giống như số cơ số 32, một chữ số chính là một ký tự trong khóa, khóa A K E Y (tương ứng với số 44217) cũng có thể viết là

$$1 \cdot 32^3 + 11 \cdot 32^2 + 5 \cdot 32^1 + 25 \cdot 32^0$$

bởi vì A là chữ cái đầu tiên, K là chữ cái thứ 11, ... Bây giờ giả sử rằng chúng ta vô tình chọn M=32 thì giá trị của k mod 32 không ảnh hưởng khi thêm vào các bội số của 32 và hàm băm của mọi khóa chỉ đơn giản là giá trị của ký tự cuối cùng của khóa ! Một hàm băm tốt nên lấy tất cả các ký tự của một khóa để tính toán. Phương pháp đơn giản nhất để bảo đảm điều này là chọn M nguyên tố.

Nhưng hoàn cảnh thông thường nhất là khi các khóa không là các số, không ngắn, chẳng hạn có thể là một chuỗi dài các ký tự. Làm sao chúng ta có thể tính hàm băm cho một khóa cỡ như V E R Y L O N G K E Y ? Trong cách mã hóa của chúng ta số này là một chuỗi dài 55 bit

1011000101100101100101100011110111000111010110010111001

hay là số

$$22.32^{10}+5.32^9+18.32^8+25.32^7+12.32^6+15.32^5+14.32^4+7.32^3+11.32^2+5.32^1+25$$

số này quá lớn đối với các hàm số học thông thường trên hầu hết các máy tính. Thực tế chúng ta có thể gặp các khóa dài hơn rất

nhiều, trong hoàn cảnh như thế chúng ta vẫn có thể tính một hàm băm giống như trên, lần lượt chuyển đổi từng mẩu một của khóa. Một lần nữa chúng ta lợi dụng tính chất của hàm mod và dùng một cách tính toán đơn giản gọi là phương pháp Horner (xem chương 36). Phương pháp này dựa trên một cách viết khác của số tương ứng với các khóa với ví dụ trên, chúng ta viết biểu thức tính khóa như sau:

(((((((((22.32+5)32+18)32+25)32+12)32+15)32+14)32+7)32+11)32+5)32+25

Điều này dựa trên một phương pháp số học trực tiếp để tính hàm băm như sau:

```
h := key[1];
for j:=2 to keysize do
   begin
      h := ((h*32)+key[j]) mod M;
   end;
```

Trong đó h là giá trị băm cần tính và key[i] được giả sử chứa j nếu ký tự thứ i trong khóa là ký tự thứ j trong bảng chữ cái. Hàm chuyển đổi ký tự ord của Pascal có thể được dùng để tính (và một bảng chữ cái lớn có thể được cung cấp nếu cần bằng cách dùng 64 hay 128 thay vì 32). Nếu không có hàm mod, đoạn chương trình trên sẽ tính số tương ứng với khóa nhỏ trong phương trình trên, nhưng tính toán có thể tràn đối với khóa dài. Tuy nhiên với sự hiện diện của hàm mod, nó tính hàm băm chính xác bởi vì các tính chất cộng và nhân của hàm mod, và mặt khác trường hợp tràn sẽ tránh khỏi bởi vì hàm mod luôn luôn cho một kết quả nhỏ hơn M. Địa chỉ băm được tính bởi chương trình này cho V E R Y L O N G K E Y với M=101 là 97.

Đối với các thuật toán sau đây, chúng ta đã giả sử rằng các khóa là nguyên và hàm băm đơn giản là mod. Điều này không có nghĩa là đề nghị sử dụng quy ước đó trong thực hành; thực ra hầu hết các ứng dụng bao gồm các cấu trúc dữ liệu với các khóa dài hơn sẽ đòi hỏi phải dùng hàm băm nói trên. Các chương trình bên dưới sẽ được sửa đổi thích hợp trong từng ứng dụng cụ thể.

XÍCH NGĂN CÁCH

Các hàm băm nói trên đổi các khóa thành các địa chỉ bảng, chúng ta cần xử lý tổng hợp hai cách áp đến cùng một địa chỉ. Phương pháp đơn giản nhất có thể nghĩ đến ngay là xây dựng một danh sách liên kết các mẩu tin mà khóa của chúng áp tới địa chỉ đó. Bởi vì các khóa mà áp dến cùng một vị trí trên bảng được lưu trong một danh sách liên kết, chúng nó cũng phải được lưu theo thứ tự. Điều này đưa đến một sự tổng quát của phương pháp tìm kiếm trên danh sách mà chúng ta đã thảo luận trong chương 14. Thay vì duy trì một danh sách đơn với một nút dẫn đầu head chúng ta lưu M danh sách với M nút dẫn đầu danh sách và chúng được khởi động như sau:

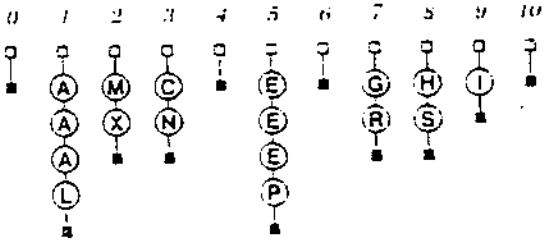
```
type link=↑node;
     node=record key,info:integer; next:link end;
var heads:array[0..M] of link; t,z:link;
procedure initialize;
   var i:integer;
   begin
   new(z); z↑.next:=z;
   for i:=0 to M-1 do
      begin new(heads[i]); heads[i]↑.next:=z end;
   end;
```

Bây giờ các thủ tục trong chương 14 có thể được sử dụng cùng với một hàm băm. Ví dụ , listinsert(v,heads[v mod M]) có thể được dùng để thêm vào bảng và t:=listsearch(v,heads[v mod M]) có thể được dùng để tìm thấy mẩu tin đầu tiên với khóa v và tiếp tục gọi t:=listsearch(v,t) cho tới khi t=z để tìm các mẩu tin kế tiếp có khóa v.

Ví dụ các khóa được chèn liên tục vào một bảng khởi tạo trống bằng cách dùng hàm băm như trong hình 16.1, thì ta sẽ được kết quả như trong hình 16.2. Phương pháp này quen được gọi là xích ngăn cách (seperate chaining) bởi vì các mẩu tin xung đột được "xích" cả lại trong các danh sách rời nhau.

key:	A	S	E	A	R	C	H	I	N	G	E	X	A	M	P	L	E
hash:	1	3	5	1	7	3	8	9	3	7	5	2	1	2	5	1	5

Hình 16.1 *Một hàm băm (M=11)*



Hình 16.2 *Xích ngăn cách.*

Hiển nhiên, thời gian cho một thao tác tìm kiếm phụ thuộc vào độ dài của danh sách và các vị trí liên quan của các khóa trong chúng. Các danh sách cũng có thể lưu trữ không thứ tự, việc duy trì các danh sách được sắp xếp sẽ không quan trọng cho áp dụng này bởi vì các danh sách khá ngắn.

Với một lần "tìm kiếm không thành công" (tìm kiếm một mẩu tin với một khóa không có trong bảng), chúng ta có thể giả sử rằng hàm băm làm cho mỗi danh sách trong M danh sách có khả năng được truy xuất như nhau và tìm kiếm tuần tự trên mỗi danh sách chỉ duyệt một nửa danh sách về mặt trung bình. Độ dài trung bình của danh sách được kiểm tra (không kể z) đối với tìm không thành công trong ví dụ này là $(0+4+2+2+0+4+0+2+2+1+0)/11 \approx 1.55$. Đây là thời gian trung bình cho một tìm kiếm không thành công khi mà các danh sách không được sắp thứ tự; khi sắp thứ tự danh sách chúng ta có thể giảm một nửa thời gian.

Với mỗi lần tìm "kiếm không thành công" (tìm kiếm một trong các mẩu tin của bảng), chúng ta giả sử rằng mỗi mẩu tin đều có khả năng được tìm thấy như nhau: bảy khóa sẽ được tìm thấy khi phần tử đầu tiên của mỗi danh sách được kiểm tra, sáu khóa sẽ được tìm thấy khi phần tử thứ hai được kiểm tra, v.v..., vì vậy trung bình là $(7.1+6.2+2.3+2.4)/17=1.94$ (sự tính toán này giả sử rằng các khóa bằng nhau được phân biệt bởi một chỉ danh duy nhất hay một cách nào đó, và chương trình tìm kiếm được sửa đổi thích hợp để có thể tìm kiếm cho mỗi khóa riêng lẻ.).

TÍNH CHẤT 16.1 *Xích ngăn cách thu hẹp số lần so sánh của tìm kiếm tuần tự xuống M lần (về mặt trung bình) và dùng bộ nhớ trợ giúp cho M liên kết.*

Bởi vì mỗi giá trị trong M giá trị băm và "có khả năng như nhau" nên do thiết kế của hàm băm nếu N khóa trong bảng lớn hơn M nhiều thì một xấp xỉ tốt độ dài trung bình của các danh sách là N/M.

Cài đặt nói trên dùng một bảng băm của các liên kết tới các nút dẫn đầu của các danh sách. Một phương pháp khác để lưu M nút dẫn đầu danh sách là khử bỏ các nút dẫn đầu và sửa heads thành một bảng các liên kết tới các khóa đầu tiên trong các danh sách. Điều này sẽ dẫn đến một vài phức tạp trong thuật toán. Ví dụ việc thêm một mẩu tin mới vào đầu của một danh sách sẽ khác với thêm vào vị trí khác của danh sách, bởi vì nó bao gồm sự sửa đổi một đầu vào trong một bảng các liên kết chứ không phải một trường của một mẩu tin. Một cài đặt khác nữa là đặt khóa đầu tiên vào bảng các liên kết. Mặc dù những cài đặt này tiết kiệm không gian lưu trữ trong một số tình huống, nhưng M thường đủ nhỏ so với N nên sự quy ước dùng các nút trợ giúp dẫn đầu danh sách có lẽ được thừa nhận.

Trong một cài đặt xích ngăn cách, M thường được chọn tương đối nhỏ để không dùng một khối lớn bộ nhớ liên tục. Nhưng có lẽ tốt nhất là chọn M đủ lớn sao cho các danh sách đủ ngắn để sự tìm kiếm tuần tự trở nên hiệu quả. Theo kinh nghiệm thì người ta có thể chọn M khoảng một phần mười số các khóa trong bảng để mỗi danh sách chứa khoảng 10 khóa. Một trong những ưu điểm của xích ngăn cách là quyết định này không nguy hại: nếu có nhiều khóa hơn mong đợi thì các quá trình tìm kiếm sẽ dài hơn một ít, nếu các khóa ít hơn thì có thể dùng phí một ít không gian lưu trữ.

Nếu bộ nhớ thực sự là một tài nguyên giới hạn, việc chọn M thích hợp sẽ cải tiến sự thực hiện được M lần.

DÒ TUYẾN TÍNH

Nếu số phần tử được đặt trong bảng băm có thể được ước lượng trước và đủ bộ nhớ liên tục để lưu tất cả các khóa thì có lẽ không

đáng để dùng các liên kết trong bảng băm. Nhiều phương pháp đề nghị lưu trữ N mẩu tin trong một bảng kích thước MN, dựa vào các nơi trống trong bảng để giải quyết sự tranh chấp. Các phương pháp như thế được gọi là các phương pháp băm địa chỉ mở (open-addressing).

Phương pháp địa chỉ mở đơn giản nhất được gọi là dò tuyến tính (linear probing): khi có một sự tranh chấp (khi chúng ta băm đến một nơi trong bảng mà đã bị chiếm và khóa ở nơi đó không bằng với khóa tìm kiếm) thì dò đến vị trí kế tiếp trong bảng, nghĩa là so sánh khóa trong mẩu tin đó với khóa tìm kiếm. Có ba khả năng trong quá trình dò: nếu các khóa bằng nhau thì quá trình tìm kết thúc thành công; nếu không có mẩu tin ở đó thì quá trình tìm kết thúc không thành công; trường hợp ngược lại dò đến vị trí kế tiếp, tiếp tục tới khi hoặc khóa tìm kiếm được thấy hoặc có một vị trí chưa dùng trong bảng. Nếu một mẩu tin chứa khóa tìm kiếm được chèn vào sau một quá trình tìm kiếm không thành công thì có thể dễ dàng đặt vào vị trí trống của bảng nơi mà quá trình tìm kết thúc. Phương pháp này dễ dàng được cài đặt như sau:

```
procedure hashinitialize;
   var i:integer;
   begin
      for i:=0 to M do a[i].key:=maxint;
   end;
function hashinsert(v:integer):integer;
   var x:integer;
   begin
   x:=h(v);
   while a[x].key<>maxint do x:=(x+1) mod M;
   a[x].key:=v;
   hashinsert:=x;
   end;
```

Dò tuyến tính đòi hỏi một giá trị khóa đặc biệt để đánh dấu một vị trí trống trong bảng: chương trình trên dùng maxint cho mục đích đó. Sự tính toán tương ứng x:= (x+1) mod M nhằm để kiểm tra vị trí kế tiếp (quay ngược trở lại từ đầu khi đến cuối của bảng). Chú ý rằng chương trình không kiểm tra bảng có khả năng bị lấp đầy hay không. (Điều gì sẽ xảy ra trong trường hợp này ?).

Hình 16.3 *Một hàm băm (M=19)*

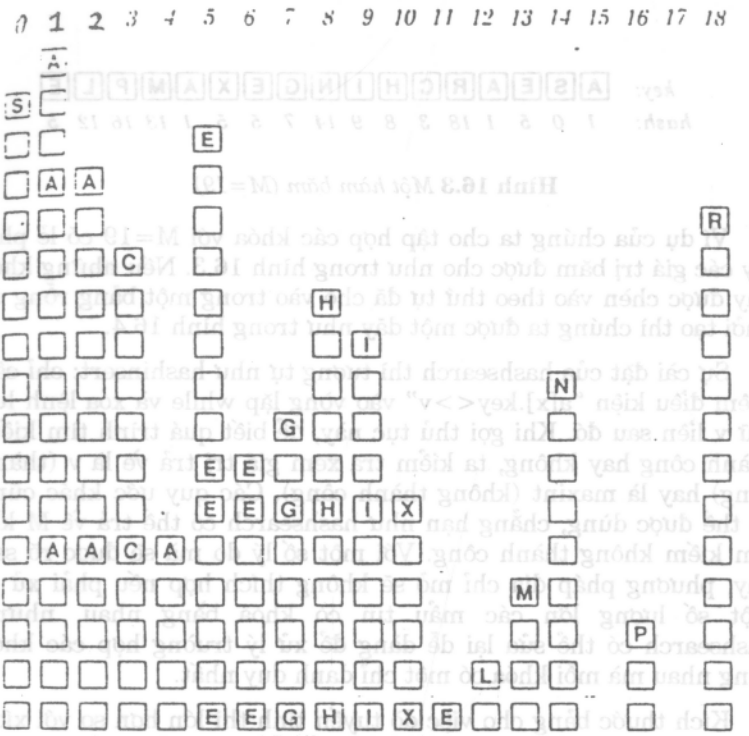
Ví dụ của chúng ta cho tập hợp các khóa với M=19 có lẻ phải lấy các giá trị băm được cho như trong hình 16.3. Nếu những khóa này được chèn vào theo thứ tự đã cho vào trong một bảng rỗng đã khởi tạo thì chúng ta được một dãy như trong hình 16.4.

Sự cài đặt của hashsearch thì tương tự như hashinsert: chỉ cần thêm điều kiện "a[x].key<>v" vào vòng lặp while và xóa lệnh lưu trữ v liền sau đó. Khi gọi thủ tục này, để biết quá trình tìm kiếm thành công hay không, ta kiểm tra xem giá trị trả về là v (thành công) hay là maxint (không thành công). Các quy ước khác cũng có thể được dùng, chẳng hạn như hashsearch có thể trả về M khi tìm kiếm không thành công. Với một số lý do mà sẽ được rõ sau này, phương pháp địa chỉ mở sẽ không thích hợp nếu phải xử lý một số lượng lớn các mẩu tin có khóa bằng nhau, nhưng hashsearch có thể sửa lại dễ dàng để xử lý trường hợp các khóa bằng nhau mà mỗi khóa có một chỉ danh duy nhất.

Kích thước bảng cho việc dò tuyến tính thì lớn hơn so với xích ngăn cách, bởi vì chúng ta phải có MN, nhưng tổng số lượng bộ nhớ cần dùng lại nhỏ hơn bởi vì không sử dụng các liên kết. số lượng trung bình các phần tử được kiểm tra cho một lần tìm kiếm thành công đối ví dụ này là 33/17 gần bằng 1.94.

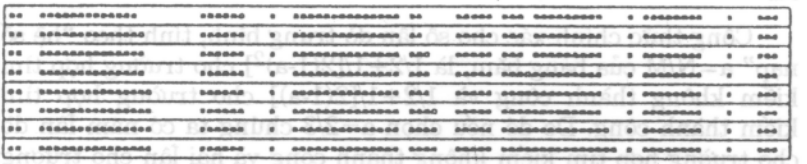
TÍNH CHẤT 16.2 *Về mặt trung bình, dò tuyến tính sử dụng ít hơn 5 lần dò đối với một bảng băm mà chưa đầy đến hai phần ba.*

Công thức chính xác cho số lần dò trung bình, tính theo "hệ số nạp" a=N/M của bảng băm, là $1/2+1/[2(1-a)^2]$ cho trường hợp tìm kiếm không thành công và $1/2+1/[2(1-a)]$ cho trường hợp tìm kiếm thành công. Do đó nếu chọn a=2/3 chúng ta có năm lần dò cho trường hợp tìm kiếm không thành công và hai lần cho trường hợp thành công. Tìm kiếm không thành công luôn đắt giá hơn tìm



Hình 16.4 *Phương pháp dò tuyến tính*

kiếm thành công và một lần tìm kiếm thành công sẽ yêu cầu ít hơn năm lần dò đến khi bảng đầy khoảng 90%. Khi bảng gần đầy (a gần bằng 1) những số này trở nên rất lớn; Điều này không nên cho xảy ra trong thực tế như chúng ta sẽ thảo luận sau này.



Hình 16.5 *Dò tuyến tính trong một bảng lớn hơn*

BĂM KÉP (DOUBLE HASHING)

Phương pháp dò tuyến tính (thực ra đối với bất kỳ phương pháp băm nào) hoạt động được bởi vì khi tìm kiếm một khóa cụ thể chúng ta duyệt qua mỗi khóa mà băm tới cùng một địa chỉ trong bảng (trường hợp đặc biệt là chính giá trị khóa ban đầu nếu nó ở trong bảng). Thật không may, trong phương pháp dò tuyến tính thì các khóa khác cũng được kiểm tra, đặc biệt khi bảng bắt đầu đầy: trong ví dụ trên, sự tìm kiếm khóa x bao gồm cả việc duyệt qua các khóa G,H,I mặc dù chúng không có cùng giá trị băm với khóa X. Thật là dở khi việc chèn một khóa cùng với một giá trị băm có thể gia tăng số lần tìm kiếm lên nhiều lần cho những khóa cùng có giá trị băm khác: trong ví dụ , việc chèn vào vị trí 17 sẽ làm gia tăng số lần tìm kiếm đối với vị trí 16. Hiện tượng này được họi là sự gom nhóm (clustering) nó có thể làm phương pháp dò tuyến tính chạy rất chậm đối với các bảng gần đầy. Hình 16.5 cho thấy sự gom nhóm trong một bảng lớn hơn.

Có một phương pháp dễ dàng để khử vấn đề gom nhóm, đó là phương pháp băm kép. Chiến lược cơ bản vẫn là như cũ; chỉ khác là thay vì kiểm tra mỗi phần tử liên tiếp tại vị trí xung đột, chúng ta dùng một hàm băm thứ hai. Điều này dễ dàng cài đặt bằng cách chèn lệnh u:= h2(v) tại vị trí bắt đầu của thủ tục và đổi lệnh x:= (x+1) mod M thành x := (x+u) mod M bên trong vòng lặp while.

Hàm băm thứ hai h2 phải được chọn cẩn thận ở một vài khía cạnh nếu không thì chương trình có thể không hoạt động tốt. Trước tiên, hiển nhiên chúng ta không muốn u=0 vì điều này sẽ dẫn tới một vòng lặp vô hạn tại vị trí xung đột. Kế đến điều rất quan trọng là M và u phải nguyên tố cùng nhau vì nếu ngược lại thì một số dãy dò tìm có thể rất ngắn (xem trường hợp M = 2u). Điều này dễ dàng ép buộc bằng cách cho M nguyên tố và u. Kế đến, hàm băm thứ hai phải khác hàm băm thứ nhất, vì nếu không sự gom nhóm phức tạp hơn có thể xuất hiện. Một hàm như thế là

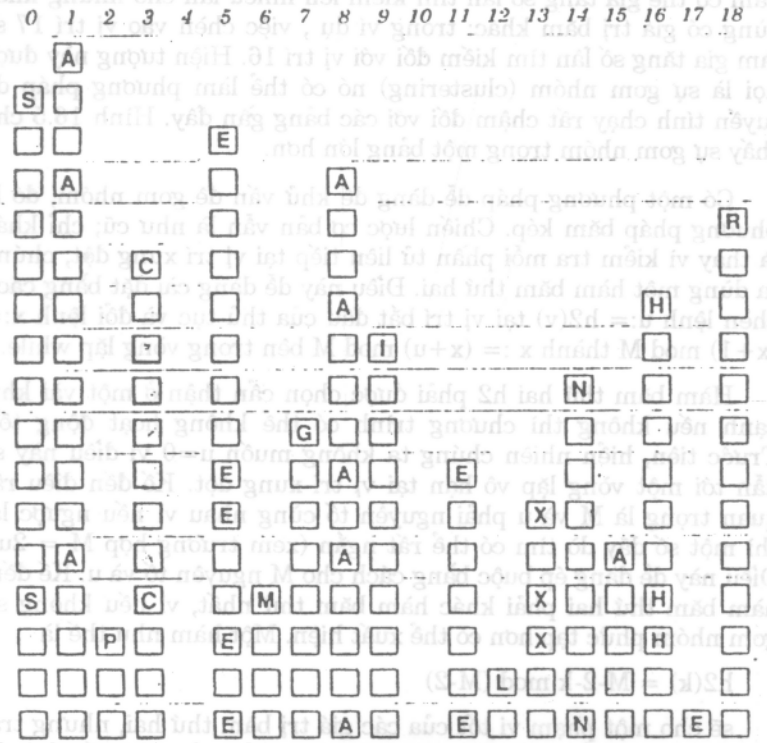
h2(k) = M-2-k mod (M-2)

sẽ cho một phạm vi tốt của các giá trị băm thứ hai, nhưng trái lại đối với những khóa dài, giá tính toán phải trả sẽ bị gấp đôi chỉ

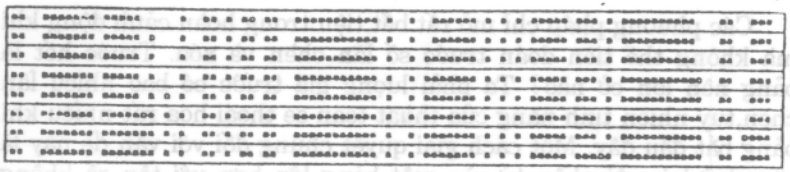
key:	A	S	E	A	R	C	H	I	N	G	E	X	A	M	P	L	E
hash 1:	1	0	5	1	18	3	8	9	14	7	5	5	1	13	16	12	5
hash 2:	7	3	3	7	6	5	8	7	2	1	3	8	7	3	8	4	3

Hình 16.6 *Một hàm băm kép (M=19)*

để tránh sự gom nhóm. Trong thực tế một hàm băm đơn giản hơn nhiều cũng đủ, ví dụ như h2(k) = 8 - (k mod 8). Hàm này chỉ dùng 3 bit cuối cùng của k; với một bảng lớn có thể dùng nhiều bit hơn, mặc dù hiệu quả, ngay cả đáng chú ý thì điều này cũng không quan trọng lắm trong thực tế.



Hình 16.7 *Băm kép*



Hình 16.8 *Băm kép trong một bảng lớn hơn*

Với các khóa mẫu của chúng ta, các hàm này sản sinh ra các giá trị băm như trong hình 16.6. Hình 16.7 cho thấy một bảng có được bằng cách chèn liên tiếp các khóa mẫu của chúng ta vào một bảng được khởi tạo trống bằng cách áp dụng băm kép với các giá trị này.

Số trung bình các phần tử được kiểm tra cho sự tìm kiếm thành công thì lớn hơn một ít so với sự dò tuyến tính cho ví dụ này: 35/17 gần bằng 2.05. Nhưng trong bảng thưa hơn, nó bị gom nhóm ít hơn như trong hình 16.8.

Với ví dụ này, số các cluster (nhóm gom) nhiều hơn hai lần so với phương pháp dò tuyến tính (hình 16.5), hay cũng vậy độ dài trung bình của cluster thì giảm đi một nữa.

TÍNH CHẤT 16.3 *Về mặt trung bình số lần dò của phương pháp băm kép ít hơn so với phương pháp dò tuyến tính .*

Công thức hiện nay để tính số lần dò trung bình của phương pháp băm kép với một hàm băm kép "độc lập" là 1/(1-a) đối với trường hợp tìm kiếm không thành công và (-ln(1-a))/a đối với trường hợp tìm kiếm thành công. (Những công thức nầy là kết quả của sự phân tích sâu về mặt toán học và chưa được kiểm chứng với a lớn). Hàm băm đơn giản dễ tính toán thứ hai đã được giới thiệu ở trên sẽ hoàn toàn không thỏa mãn điều này, nhưng nó chặt chẽ hơn, đặc biệt nếu dùng đủ bit sao cho phạm vi của các giá trị có thể gần tới M. Về mặt thực hành, điều này có nghĩa là có thể dùng một bảng nhỏ hơn để phương pháp băm kép dò tuyến tính cùng số lần tìm kiếm đối với một ứng dụng đã cho: số lần dò trung bình thì nhỏ hơn 5 đối với một lần tìm kiếm không thành công nếu bảng chưa đầy tới 80%, và đối với một lần tìm kiếm thành công nếu chưa đầy tới bảng 90%.

Các phương pháp chỉ mở rất bất tiện trong hoàn cảnh động khi mà không thể tiên đoán trước số lần chèn và xóa. Trước hết là bảng nên lớn cỡ nào? Ta phải lượng giá trước có bao nhiêu lần chèn,tuy nhiên tính năng của thuật toán sẽ thoái hóa rất nhiều khi bảng bắt đầu đầy. Một cách giải quyết chung đối với vấn đề này là băm trở lại mỗi phần tử vào một bảng lớn hơn với tần số không thường xuyên. Kế đến là một mẩu tin không thể được xóa một cách đơn giản từ một bảng được xây dựng với phương pháp dò tuyến tính hay phương pháp băm kép. Lý do là các lần chèn mẩ tin vào trong bảng sau này có thể vượt khỏi mẩu tin bị xóa, và sự tìm kiếm những mẩu tin này sẽ kết thúc tại nơi được chiếm bởi mẩu tin bị xóa. Một cách để giải quyết vấn đề nầy là dùng một khóa đặc biệt khác cho một vị trí trống, và chú ý đến những khóa nầy trong quá trình tìm kiếm. Chú ý rằng cả kích thước bảng và sự xóa mẩu tin đều không phải là vấn đề đặc biệt đối với các xích ngăn cách (Separate chaining).

BÀN LUẬN THÊM

Các phương pháp thảo luận trên đã được phân tích hoàn chỉnh và nó có thể so sánh tính năng của chúng trong một vài khía cạnh. Các công thức được cho ở trên được tóm tắt từ các phân tích chi tiết của D.E.Knuth trong quyển sách về Sắp xếp và Tìm kiếm (Sorting and Searching) của ông ta. Các công thức chỉ ra sự thoái hóa về tích năng của phương pháp địa chỉ mở khi a dần tới 1. Với M và N lớn và khi một bảng đầy ở 90% thì phương pháp dò tuyến tính sẽ đòi hỏi 50 lần dò đối với một trường hợp tìm kiếm không thành công, và đối với phương pháp băm kép thì đòi hỏi 10 lần dò. Nhưng trong thực tế, người ta khôngbao giờ để bảng băm đầy tới 90%! Với các hệ số nạp (load factor) nhỏ, chỉ cần một vài lần dò, nếu không thể dàn xếp để cho các hệ số nạp nhỏ thì không nên dùng các phương pháp băm.

So sánh phương pháp dò tuyến tính và phương pháp băm kép đối với xích ngăn cách thì phức tạp hơn, bởi vì các phương pháp địa chỉ mở dùng ít bộ nhớ hơn do không có các liên kết. Giá trị a được hiệu chỉnh dựa vào kích thước tương đối giữa các khóa và các liên kết. Điều này nghĩa là không chỉ đơn giản chọn phương pháp

xích tách biệt để giảm số lần tìm kiếm nếu như số mẩu tin được xử lý không biết trước và được cấp phát bộ nhớ đôi dào và dùng phương pháp băm kép để tìm kiếm một tập hợp khóa mà kích thước của chúng có thể được tiên đoán trước thời hạn.

Nhiều phương pháp băm khác được phát biểu và có áp dụng trong vài hoàn cảnh đặc biệt. Mặc dù chúng ta không thể đi sâu vào chi tiết, chúng ta sẽ xem tóm tắt hai ví dụ để minh họa bản chất của các phương pháp băm đặc biệt. Hai phương pháp này và nhiều phương pháp khác được mô tả đầy đủ trong quyển sách của Knuth và Gonnet.

Trước hết, là phương pháp băm thứ tự (odered hashing), nó khai thác thứ tự trong một bảng địa mở. Trong phương pháp dò tuyến tính chuẩn, chúng ta dừng khi tìm thấy một vị trí trống hay một mẩu tin có khóa bằng với khóa đang tìm kiếm; Trong phương pháp băm thứ tự, chúng ta dừng quá trình tìm kiếm khi tìm thấy một mẩu tin có khóa lớn hơn hay bằng với khóa đang tìm kiếm (bảng phải được xây dựng khôn khéo để có thể làm được như thế). Phương pháp này làm cho thời gian tìm kiếm không thành công xấp xỉ với thời gian tìm kiếm thành công. (Điều này giống như sự cải tiến trong phương pháp xích ngăn cách.) Phương pháp này có ích đối với các ứng dụng mà trường hợp tìm kiếm không thành công thường xuyên xảy ra. Ví dụ, một hệ thống xử lý văn bản có thể có một thuật toán cho các từ nối mà hoạt động tốt cho hầu hết các từ ngoại trừ các trường hợp quái dị. Tình thế có thể được xử lý bằng cách quan sát tất cả các từ mà phải được xử lý bằng một phương pháp đặc biệt để thỏa mãn hầu hết là xảy ra trường hợp tìm kiếm không thành công.

Tương tự, có các phương pháp tìm kiếm di chuyển một vài mẩu tin trong suốt quá trình tìm kiếm không thành công để cho sự tìm kiếm thành công có hiệu quả hơn. Thực ra thì R. P. Brent đã phát triển một phương pháp mà thời gian tìm kiếm trung bình cho một trường hợp tìm kiếm không thành công có thể bị chận bởi một hằng số, đây là một phương pháp rất hữu dụng cho các ứng dụng mà trường hợp tìm kiếm không thành công thường xuyên xảy ra trong một bảng rất lớn cỡ như các tự điển.

Đây chỉ là hai ví dụ trong một số lớn các thuật toán được cải tiến cho các phương pháp băm. Hiện nay có nhiều cải tiến rất thú vị và có các ứng dụng quan trọng. Tuy nhiên, phương pháp xích ngăn cách và băm kép thì lại đơn giản, hiệu quả và hoàn toàn thích hợp cho hầu hết các ứng dụng.

Các phương pháp băm được thích hơn so với các cấu trúc cây nhị phân trong các chương trước đối với nhiều ứng dụng bởi vì nó đơn giản hơn và lại cho thời gian tìm kiếm nhanh hơn nếu như chấp nhận tốn chỗ cho một bảng lớn. Các cấu trúc cây nhị phân có thuận lợi nhờ vào tính động của chúng (không cần biết trước số lần chèn vào cây), chúng cũng có thể bảo đảm trong trường hợp xấu nhất (trong khi mỗi phần tử đều có thể băm tới cùng một nơi ngay cả đối với phương pháp băm tốt nhất), và chúng cung cấp một phạm vi thao tác rất rộng lớn (quan trọng nhất là chức năng sắp xếp). Khi các đối tượng vừa nói trên không quan trọng thì chắc chắn phương pháp băm là phương pháp tìm kiếm được lựa chọn.

BÀI TẬP

1. Mô tả làm thế nào để cài đặt một hàm băm bằng cách dùng một một bộ phát sinh số ngẫu nhiên tốt. Có ý nghĩa hay không nếu cài đặt một bộ phát sinh ngẫu nhiên bằng cách dùng một hàm băm.

2. Lượng giá trường hợp xấu nhất khi chèn N khóa vào một bảng được khởi tạo trống bằng cách dùng xích ngăn cách với các danh sách không thứ tự.

3. Cho biết nội dung của bảng băm có được khi chèn các khóa E A S Y Q U E S T I O N theo thứ tự đó vào một bảng được khởi tạo trống kích thước 13 bằng phương pháp dò tuyến tính.

4. Cho biết nội dung của bảng băm có được khi chèn các khóa E A S Y Q U E S T I O N theo thứ tự đó vào một bảng được khởi tạo trống kích thước 13 bằng phương pháp băm kép. (Trong đó h1(k) lấy từ câu hỏi trước, h2(k) = 1 + (k mod 11).)

5. Cần khoảng bao nhiêu lần dò khi dùng phương pháp băm kép để xây dựng một bảng với N khóa bằng nhau?

6. Bạn nên dùng phương pháp băm nào cho một ứng dụng mà có nhiều trường hợp khóa trùng nhau?

7. Giả sử rằng cho biết trước số phần tử sẽ được đặt vào bảng băm. Với những điều kiện nào thì phương pháp xích ngăn cách thích hợp hơn phương pháp băm kép.

8. Giả sử một lập trình vfiên có một lỗi trong chương trình dùng phương pháp băm kép mà một trong các hàm luôn trả về cùng một giá trị (khác 0). Mô tả điều gì sẽ xảy ra trong mỗi tình huống (khi hàm thứ nhất bị sai và khi hàm thứ hai bị sai).

9. Nên dùng hàm băm nào nếu biết trước rằng các giá trị khóa rơi vào một phạm vi tương đối nhỏ.

10. Phê bình thuật toán sau đây, thuật toán nầy nhằm mục đích xóa khóa khỏi một bảng băm được xây dựng bằng phương pháp dò tuyến tính. Quét qua phải kể từ phần tử được xóa để ra một vị trí trống, kế đến quét trái để tìm một phần tử có cùng giá trị băm, sau cùng thay thế phần tử được xóa bởi phần tử vừa tìm được.

17
TÌM KIẾM DỰA VÀO CƠ SỐ (RADIX SEARCHING)

Nhiều phương pháp tìm kiếm chỉ kiểm tra các khóa tìm kiếm theo từng bit thay vì phải so sánh toàn bộ các khóa với nhau. Những phương pháp này được gọi là các phương pháp tìm kiếm dựa vào cơ số, chúng làm việc với các bit của khóa, ngược lại với các phương pháp hash là dùng một bảng chuyển đổi của các khóa. Giống như các phương pháp sắp xếp dựa vào cơ số (xem chương 10), những phương pháp này hữu dụng khi các bit của các khóa tìm kiếm được truy xuất dễ dàng và các giá trị của các khóa tìm kiếm được phân bố tốt.

Thuận lợi chính của các phương pháp tìm kiếm dựa vào cơ số là chúng cung cấp một phương pháp dễ dàng xử lý các khóa có chiều dài thay đổi; một số trường hợp thì tiết kiệm không gian lưu trữ bằng cách lưu trữ bộ phận khóa bên trong cấu trúc dữ liệu; ngoài ra chúng có thể truy xuất dữ liệu nhanh có thể sánh được với cả cây tìm kiếm nhị phân và phương pháp hash. Sự bất lợi của phương pháp này là nếu dữ liệu không đối xứng thì có thể dẫn tới các cây thoái hóa với tính năng xấu (và dữ liệu bao gồm các ký tự thì lại không đối xứng) và một số trường hợp thì bị hao phí không gian lưu trữ. Cũng như những phương pháp sắp xếp cơ số, các phương pháp tìm kiếm dựa cơ số được thiết kế để tận dụng đặc trưng của kỹ thuật máy tính: bởi vì chúng dùng tính chất số của các khóa, rất khó hay thậm chí không thể cài đặt chúng một cách hiệu quả trong các ngôn ngữ như Pascal.

Chúng ta sẽ điểm qua một chuỗi các phương pháp, mỗi phương pháp hiệu chỉnh một vấn đề còn tồn động trong phương pháp trước, cuối cùng là một phương pháp quan trọng mà hoàn toàn

hữu dụng cho các ứng dụng tìm kiếm có chứa các khóa dài. Hơn nữa chúng ta sẽ xem sự tương tự của nó với phương pháp "sắp xếp thời gian tuyến tính" ("linear-time sort") của chương 10, đó là phương pháp tìm kiếm "thời gian hằng" ("Cont stant-time") được dựa vào quy luật tương tự.

CÁC CÂY TÌM KIẾM SỐ HỌC (Digital Search Trees)

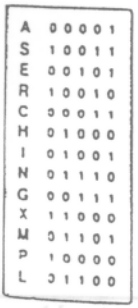
Phương pháp tìm kiếm dựa vào cơ số đơn giản nhất là tìm kiếm trên cây số học (digital tree): thuật toán giống hệt như tìm kiếm trên cây nhị phân, ngoại trừ sự phân nhánh cây không tương ứng với kết quả của sự so sánh các khóa mà tương ứng với các bit của khóa. Ở tầng đầu tiên thì bit dẫn đầu được dùng, ở tầng thứ hai thì bit thứ hai được dùng, và tiếp tục tới khi gặp một nút ngoài. Chương trình làm thao tác nói trên rất giống với chương trình tìm kiếm trên cây nhị phân, chỉ khác là sự so sánh khóa được thay thế bằng cách gọi hàm bits mà chúng ta đã dùng trong phương pháp sắp xếp dựa vào cơ số. (Nhắc lại từ chương 10 là:

$$bit(x, k,j) = (x \textbf{ div } 2^k) \textbf{ mod } 2^j;$$

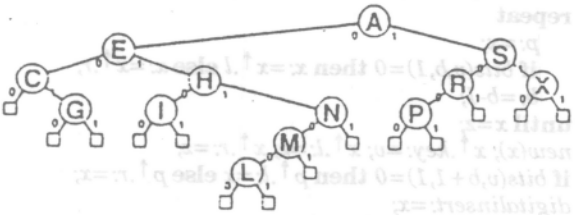
· hàm này có thể cài đặt hiệu quả bởi các thao tác trên bit như dịch chuyển, and, or...).

```
function digitalsearch(v:integer; x:link):link;
  var b:integer;
  begin
  z↑.key := v; b:=maxb;
  repeat
    if bits(v,b,1)=0 then x:=x↑.l else x:=x↑.r;
    b:=b-1;
  until v=x↑.key;
  digitalsearch:=x;
  end;
```

Cấu trúc dữ liệu cho chương này thì giống như chúng ta đã dùng cho cây nhị phân tìm kiếm. Hằng số maxb là số bit trong các khóa được sắp xếp. Chương trình giả sử rằng bit đầu tiên trong mỗi khóa (bit thứ maxb + 1 từ bên phải) là 0 (có thể là kết quả có được khi gọi hàm bits với đối số thứ ba là maxb), sao cho sự tìm



A	00001
S	10011
E	00101
R	10010
C	00011
H	01000
I	01001
N	01110
G	00111
X	11000
M	01101
P	10000
L	01100



Hình 17.1 *Một cây tìm kiếm số học*

kiếm kết thúc bằng cách đặt x:=digitalsearch(v,head), trong đó head là một liên kết đến nút header của cây (tree header node) có khóa 0 và có liên kết trái trỏ tới cây tìm kiếm. Do đó thủ tục khởi động cho chương trình này thì giống như cho tìm kiếm trên cây nhị phân, ngoại trừ chúng ta bắt đầu với head↑.l:=z thay vì head↑.r:=z.

Chúng ta đã thấy trong chương 10 là các khóa bằng nhau thì bị đại kị trong phương pháp sắp xếp dựa vào cơ số, không những trong thuật toán cụ thể nầy mà còn trong các thuật toán mà chúng ta sẽ nghiên cứu sau này. Do đó trong chương này chúng ta sẽ giả sử rằng tất cả các khóa xuất hiện trong cấu trúc dữ liệu là phân biệt nhau: nếu cần, có thể dùng một danh sách liên kết cho mỗi giá trị khóa của các mẩu tin mà giá trị khóa của chúng bằng nhau. Như trong các chương trước, chúng ta sẽ giả sử rằng chữ thứ i của bảng chữ cái được biểu diễn bởi biểu diễn nhị phân 5 bit của i. Các khóa mẫu dùng trong chương này được cho trong hình 17.1. Để phù hợp với hàm bits, chúng ta xem các bit được đánh số từ 0 đến 4 theo thứ tự từ phải tới trái. Do đó bits 0 là bits duy nhất của A mà khác 0 và bit 4 là bit duy nhất của p mà khác 0.

Thủ tục chèn vào cây tìm kiếm số cũng được kế thừa trực tiếp từ thủ tục tương ứng cho cây tìm kiếm nhị phân:

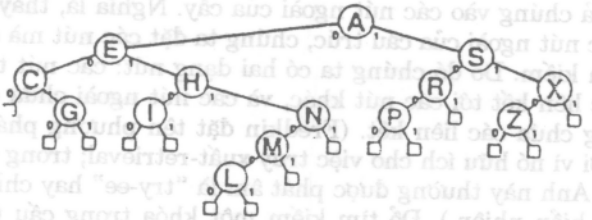
```
function digitalinsert(v:integer; x:link):link;
   var p:link; b:integer;
   begin
   b:=maxb;
   repeat
      p:=x;
      if bits(v,b,1)=0 then x:=x↑.l else x:=x↑.r;
      b:=b-1;
   until x=z;
   new(x); x↑.key:=v; x↑.l:=z; x↑.r:=z;
   if bits(v,b+1,1)=0 then p↑.l:=x else p↑.r:=x;
   digitalinsert:=x;
   end;
```

Cây được xây dựng bởi chương trình này khi chúng ta chèn các khóa mẫu vào một cây được khởi tạo trống như trong hình 17.1. Hình 17.2 cho thấy điều gì xảy ra khi một khóa mới z=11010 được thêm vào cây trong hình 17.1. Chúng ta đến phía phải hai lần bởi vì hai bit dẫn đầu của z là 1 và kế đến chúng ta chuyển qua trái, ở đó chúng ta chạm phải nút ngoài ở bên trái của x và z được chèn vào cây.

Trường hợp xấu nhất cho các cây được xây dựng với phương pháp tìm kiếm số học thì tốt hơn nhiều so với cây tìm kiếm nhị phân, nếu số lượng khóa lớn và các khóa thì không dài. Chiều dài của con đường dài nhất trong một cây tìm kiếm số học là chiều dài của bit dẫn đầu dài nhất của hai khóa bất kỳ trong cây mà có các bit dẫn đầu trùng nhau, và con số này thì tương đối nhỏ đối với nhiều ứng dụng (ví dụ, nếu các khóa bao gồm các bit ngẫu nhiên).

TÍNH CHẤT 17.1 *Về mặt trung bình đối với một cây tìm kiếm số học được xây dựng từ các khóa b-bit ngẫu nhiên thì một thao tác tìm kiếm, hay chèn vào cây sẽ đòi hỏi khoảng lgN phép so sánh và b phép so sánh trong trường hợp xấu nhất.*

Hiển nhiên rằng không bao giờ có con đường dài hơn số bit của các khóa: ví dụ, một cây tìm kiếm số được xây dựng từ các khóa 8 ký tự với 6 bit mỗi ký tự sẽ không có con đường dài hơn 48, ngay cả nếu có hàng trăm ngàn khóa, về mặt trung bình các cây tìm kiếm số học thì gần như cân bằng hoàn toàn, kết quả này đòi hỏi một sự phân tích vượt khỏi phạm vi của quyển sách này, mặc dù



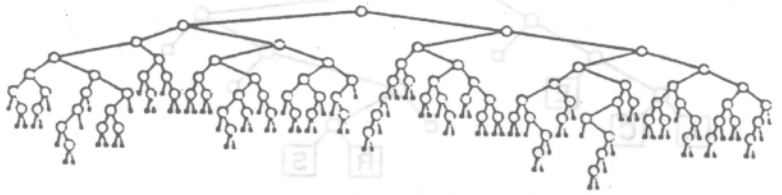
Hình 17.2 *Chèn (Z) vào một cây tìm kiếm số học*

nó đúng với trực giác đơn giản là bit "kế tiếp" của một khóa ngẫu nhiên thì khả năng là 0 hay 1 là như nhau, vì vậy một nửa sẽ rơi vào một cạnh của nút bất kỳ. Hình 17.3 cho thấy một cây tìm kiếm số học được xây dựng từ 95 khóa 7 bit ngẫu nhiên, cây này thì hoàn toàn cân bằng.

Do đó các cây tìm kiếm số học cho ta một lựa chọn hấp dẫn hơn so với cây tìm kiếm nhị phân chuẩn, sự trích ra các bit thì dễ dàng thực hiện cũng như so sánh khóa.

TRIE TÌM KIẾM CƠ SỐ

Rất thường xảy ra trường hợp mà các khóa tìm kiếm rất dài, có thể bao gồm 20 ký tự hay dài hơn. Trong tình huống như thế, giá phải trả của sự so sánh một khóa tìm kiếm với một khóa từ cấu trúc dữ liệu có thể rất đáng kể đến nỗi không thể bỏ sót.Cây tìm kiếm số học dùng phép so sánh tại mỗi nút của cây; trong phần này chúng ta sẽ thấy rằng trong hầu hết các trường hợp chỉ cần một phép so sánh cho mỗi lần tìm kiếm.

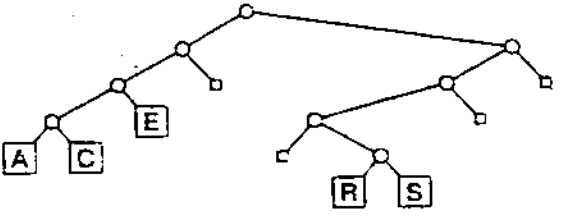


Hình 17.3 *Một cây tìm kiếm số học lớn*

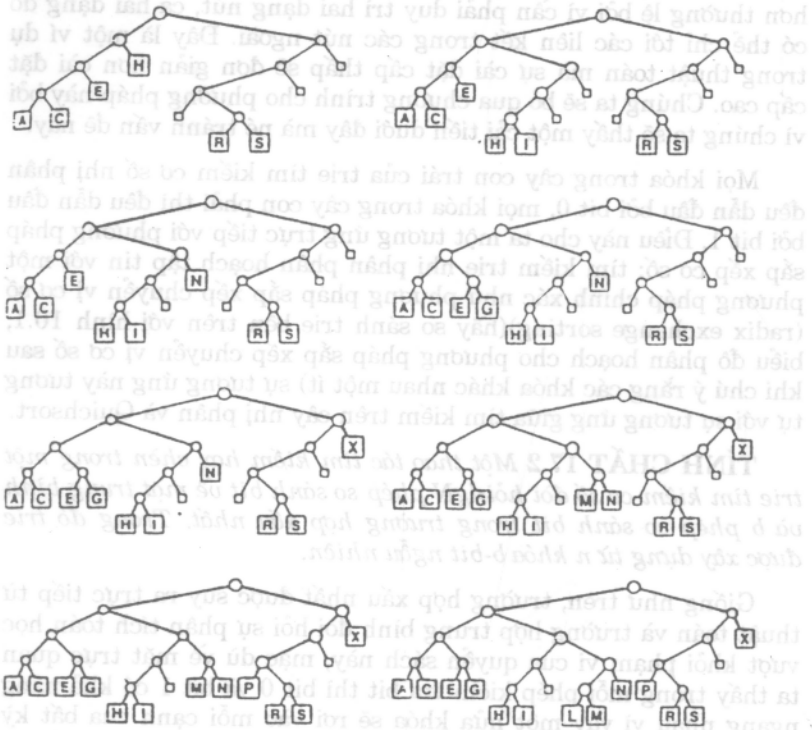
Ý tưởng là không lưu trữ các khóa trong các nút của cây, mà đặt tất cả chúng vào các nút ngoài của cây. Nghĩa là, thay vì dùng z cho các nút ngoài của cấu trúc, chúng ta đặt các nút mà chứa các khóa tìm kiếm. Do đó chúng ta có hai dạng nút: các nút trong chỉ chứa các liên kết tới các nút khác, và các nút ngoài chứa các khóa và không chứa các liên kết. (Fredkin đặt tên phương pháp này là "trie" bởi vì nó hữu ích cho việc truy xuất-retrieval; trong đối thoại từ tiếng Anh này thường được phát âm là "try-ee" hay chỉ là "try" với lý lẻ hiển nhiên.). Để tìm kiếm một khóa trong cấu trúc như thế, chúng ta chỉ phân nhánh tương ứng với các bit của nó như trên, nhưng chúng ta không so sánh nó với bất kỳ một phần tử nào đến khi gặp một nút ngoài. Mỗi khóa trong cây được lưu trữ trong một nút ngoài trên con đường được mô tả bởi mẫu bit dẫn đầu của khóa và mỗi khóa tìm kiếm dừng lại ở một nút ngoài, một phép so sánh toàn bộ được thực hiện để kết thúc quá trình tìm kiếm.

Hình 17.4 cho thấy một trie tìm kiếm cơ số nhị phân cho các khóa A S E R C. Ví dụ do ba bit đầu tiên của E là 001; nhưng bởi vì không có khóa nào trong trie bắt đầu với dãy bit 101 nên để chạm tới E chúng ta khởi đầu từ gốc và đến trái, trái, phải nên khi đi theo phải, trái, phải ta sẽ gặp một nút ngoài. Trước khi nghĩ về cách chèn vào cây, độc giả nên suy gẫm về tính chất rất ngạc nhiên của cấu trúc trie là nó độc lập với thứ tự chèn các khóa vào cây: có một trie duy nhất với một tập hợp khóa phân biệt được cho trước.

Như thường lệ, sau một quá trình tìm kiếm không thành công, chúng ta có thể chèn khóa tìm kiếm bằng cách thay nút ngoài tại nơi kết thúc quá trình tìm kiếm với điều kiện là nó không chứa



Hình 17.4 *Một trie tìm kiếm cơ số*



Hình 17.5 *Xây dựng một trie tìm kiếm cơ số*

một khóa nào. Đây là trường hợp khi H được chèn vào trie trong hình 17.4 và kết quả như trie đầu tiên tronh hình 17.5. Nếu nút ngoài kết thúc quá trình tìm kiếm có chứa một khóa, khi đó phải thay thế nó bởi nút ngoài mà sẽ chứa khóa tìm kiếm và kết thúc quá trình tìm kiếm. Thật không may mắn nếu các khóa này trùng nhau ở các vị trí bit nhiều hơn thì phải cần thêm một vài nút ngoài mà không chứa khóa (hay một vài nút ngoài có chứa nút con ngoài trống). Điều này xảy ra khi chèn I vào cây như trong hình 17.5. Phần còn lại của hình 17.5 cho thấy đầy đủ ví dụ của chúng ta khi các khóa N G X M P L được thêm vào.

Sự cài đặt phương pháp này bằng ngôn ngữ Pascal thì phức tạp hơn thường lệ bởi vì cần phải duy trì hai dạng nút, cả hai dạng đó có thể chỉ tới các liên kết trong các nút ngoài. Đây là một ví dụ trong thuật toán mà sự cài đặt cấp thấp sẽ đơn giản hơn cài đặt cấp cao. Chúng ta sẽ bỏ qua chương trình cho phương pháp này bởi vì chúng ta sẽ thấy một cải tiến dưới đây mà né tránh vấn đề này.

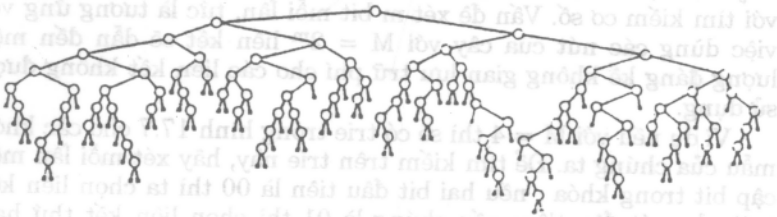
Mọi khóa trong cây con trái của trie tìm kiếm cơ số nhị phân đều dẫn đầu bởi bit 0, mọi khóa trong cây con phải thì đều dẫn đầu bởi bit 1. Điều này cho ta một tương ứng trực tiếp với phương pháp sắp xếp cơ số: tìm kiếm trie nhị phân phân hoạch tập tin với một phương pháp chính xác như phương pháp sắp xếp chuyển vị cơ số (radix exchange sorting)(hãy so sánh trie bên trên với hình 10.1, biểu đồ phân hoạch cho phương pháp sắp xếp chuyển vị cơ số sau khi chú ý rằng các khóa khác nhau một ít) sự tương ứng này tương tự với sự tương ứng giữa tìm kiếm trên cây nhị phân và Quichsort.

TÍNH CHẤT 17.2 *Một thao tác tìm kiếm hay chèn trong một trie tìm kiếm cơ số đòi hỏi lgN phép so sánh bit về mặt trung bình và b phép so sánh bit trong trường hợp xấu nhất. Trong đó trie được xây dựng từ n khóa b-bit ngẫu nhiên.*

Giống như trên, trường hợp xấu nhất được suy ra trực tiếp từ thuật toán và trường hợp trung bình đòi hỏi sự phân tích toán học vượt khỏi phạm vi của quyển sách này, mặc dù về mặt trực quan ta thấy trong mỗi phép kiểm tra bit thì bit 0 và bit 1 có khả năng ngang nhau vì vậy một nửa khóa sẽ rơi vào mỗi cạnh của bất kỳ nút nào của trie.

Một tính chất cần quan tâm của các trie cơ số và người ta phải phân biệt chúng với cây tìm kiếm dạng khác là sự phân nhánh "một đường" (one way) đòi hỏi các khóa có chung một số lớn các bit. Ví dụ các khóa chỉ khác nhau bit cuối cùng sẽ đòi hỏi một con đường có chiều dài bằng với chiều dài khóa, bất chấp có bao nhiêu khóa ở trong cây. Đôi khi số nút trong có thể lớn hơn số lượng khóa.

TÍNH CHẤT 17.3 *Trie tìm kiếm cơ số được xây dựng từ N khóa b-bit ngẫu nhiên có khoảng N/ln2 (gần bằng 1.44N) nút về mặt trung bình.*



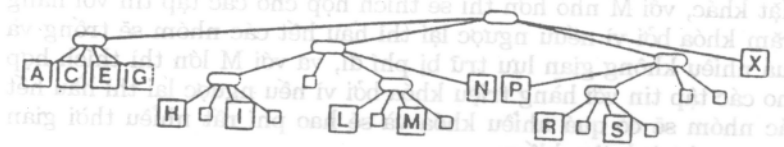
Hình 17.6 *Một trie tìm kiếm cơ số lớn*

Một lần nữa chứng minh của kết quả này hoàn toàn vượt khỏi phạm vi của quyển sách này mặc dù nó dể dàng được kiểm chứng một cách trực quan. Hình 17.6 cho thấy một trie được xây dựng từ 95 khóa 10-bit ngẫu nhiên mà có 131 nút.

Độ cao của các trie vẫn còn bị giới hạn bởi số bit trong khóa, nhưng chúng ta sẽ khảo sát khả năng xử lý mẩu tin với các khóa rất dài (chẳng hạn 1000 bit hay nhiều hơn), ví dụ như đối với dữ liệu ký tự bị mã hóa. Một phương pháp để làm ngắn các đường đi trong cây là dùng nhiều hơn hai liên kết cho mỗi nút (mặc dù điều này lại vi phạm vấn đề không gian lưu trữ bởi vì dùng quá nhiều nút); Một phương pháp khác là chuyển các con đường chứa các nhánh "một đường" (one-way) thành các liên kết đơn. Chúng ta sẽ thảo luận các phương pháp này trong hai phần tới.

TÌM KIẾM CƠ SỐ ĐA HƯỚNG

Với phương pháp sắp xếp cơ số, chúng ta thấy rằng có thể cải tiến về mặt tốc độ bằng cách xét mỗi lần nhiều hơn một bit. Điều này cũng y hệt như tìm kiếm cơ số: bằng cách kiểm tra m bit mỗi lần chúng ta có thể tăng tốc độ tìm kiếm lên một hệ số bằng 2^m.



Hình 17.7 *Một trie cơ số 4-hướng*

Tuy nhiên, có một trở ngại cần chú ý khi áp dụng ý tưởng này đối với tìm kiếm cơ số. Vấn đề xét m bit mỗi lần, tức là tương ứng với việc dùng các nút của cây với $M = 2^m$ liên kết sẽ dẫn đến một lượng đáng kể không gian lưu trữ phí cho các liên kết không được sử dụng.

Ví dụ nếu với M = 4 thì sẽ có trie trong hình 17.7 cho các khóa mẫu của chúng ta. Để tìm kiếm trên trie này, hãy xét mỗi lần một cặp bit trong khóa ; nếu hai bit đầu tiên là 00 thì ta chọn liên kết trái của nút đầu tiên; nếu chúng là 01 thì chọn liên kết thứ hai; nếu chúng là 10 thì chọn liên kết thứ ba; nếu chúng là 11 thì chọn liên kết phải. Khi đó nhánh trên tầng kế tiếp tương ứng với bit thứ ba và thứ tư, ví dụ để tìm kiếm T = 10100 trong trie của hình 17.7 ta chọn liên kết thứ ba từ gốc và kế đến liên kết thứ ba từ con thứ ba của gốc để truy xuất tới một nút ngoài, và đến đó thì quá trình tìm kiếm kết thúc không thành công. Để chèn T vào thì nút đó có thể được thay bởi nút mới chứa T (và bốn liên kết ngoài).

Chú ý rằng có một vài không gian phí trong cây này bởi vì một số lớn các liên kết ngoài không dùng. Khi M càng lớn, hiệu ứng này càng xấu thêm: số liên kết không dùng là khoảng MN/lgM đối với các khóa ngẫu nhiên. Xét về mặt khác thì đây lại là phương pháp tìm kiếm rất hiệu quả: thời gian chạy khoảng lgN. Chúng ta có thể dung hòa giữa sự hiệu quả thời gian của các trie đa hướng và hiệu quả không gian của các phương pháp khác bằng cách dùng một phương pháp lai với một giá trị M lớn ở đỉnh (ví dụ hai tầng đầu) và một giá trị nhỏ của M (hay thay bằng một phương pháp cơ bản nào đó) ở đáy. Tuy nhiên một lần nữa sự cài đặt hiệu quả của các phương pháp như thế thì hoàn toàn phức tạp bởi vì các dạng nút bội.

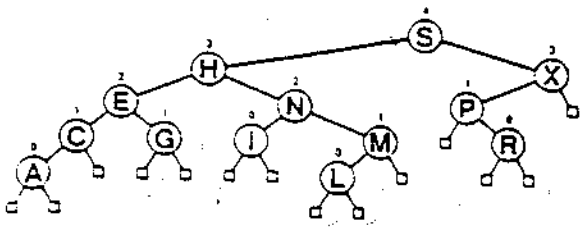
Ví dụ, một cây hai tầng 32-hướng sẽ chia tập các khóa thành 1024 nhóm, mỗi nhóm có thể được truy xuất bởi hai bước hướng xuống phía dưới cây. Điều này rất hữu dụng đối với những tập tin có hàng ngàn khóa bởi vì gần như chỉ có một vài khóa mỗi nhóm. Mặt khác, với M nhỏ hơn thì sẽ thích hợp cho các tập tin với hàng trăm khóa bởi vì nếuu ngược lại thì hầu hết các nhóm sẽ trống và quá nhiều không gian lưu trữ bị phí đi, và với M lớn thì thích hợp cho các tập tin với hàng triệu khóa bởi vì nếu ngược lại thì hầu hết các nhóm sẽ có quá nhiều khóa và sẽ hao phí rất nhiều thời gian trong quá trình tìm kiếm.

Rất thú vị nếu chú ý rằng phương pháp tìm kiếm "lai" tương ứng rất gần với phương pháp mà con người tìm kiếm đồ vật, ví dụ tìm kiếm tên trong một quyển danh bạ điện thoại. Bước đầu tiên là quyết định đa hướng ("Hãy xem, nó bắt đầu với chữ A"), kế đến có thể là các quyết định có hai hướng ("Nó đứng trước Andrens nhưng lại sau Aitken"), kế đến là tìm kiếm tuần tự ("Algonquin ... Algren, không Algorithms không được liệt kê ở đây"). Dĩ nhiên, các máy tính tốt hơn con người ở việc tìm kiếm đa hướng, vì vậy hai tầng là thích hợp. Cũng vậy, nhánh 26-hướng (ngay cả với nhiều tầng) sẽ là một thay thế thích hợp khi xét các khóa chỉ gồm các ký tự (ví dụ trong một tự điển).

Trong chương kế tiếp chúng ta sẽ thấy một phương pháp có hệ thống để cài đặt các cấu trúc mà lợi dụng phương pháp tìm kiếm cơ số cho các tập tin có kích thước tùy ý.

PATRICIA

Phương pháp tìm kiếm trên trie cơ số được phác họa ở trên có hai khuyết điểm cần chú ý: "Sự phân nhánh một hướng" sẽ dẫn đến việc tạo lập các nút ngoài trong cây, và có hai dạng nút khác nhau trong cây, điều này gây phức tạp khi thảo chương trình (đặc biệt là chương trình thêm nút mới vào trie). GD.R.Morrison đã khám phá ra một phương pháp để tránh cả hai vấn đề này, và ông đã đặt tên nó là PATRICIA (viết tắt của Practiccal Algorithm To Retrieve Information Coded In Alphanumeric). Thuật toán dưới đây không hoàn toàn giống như được trình bày bởi Morrison bởi vì ông ta thích thú trong các ứng dụng "Tìm kiếm chuỗi" (string searching) như chúng ta sẽ thấy trong chương 19. Trong ngữ cảnh hiện tại PATRICIA cho phép tìm kiếm trên N khóa dài tùy ý trên một cây



Hình 17.8 *Một cây Patricia*

có N nút nhưng lại đòi hỏi chỉ một lần so sánh toàn bộ khóa cho mỗi lần tìm kiếm.

Để tìm kiếm trên cây này, chúng ta bắt đầu từ gốc và duyệt xuống phía dưới cây, sử dụng chỉ mục bit trong mỗi nút để xem cần phải kiểm tra bit nào trong khóa tìm kiếm, chúng ta rẻ phải nếu bit là 1 và rẻ trái nếu nó là 0. Các khóa trong nút thì không được kiểm tra khi ta duyệt hướng xuống phía dưới cây. Cuối cùng chạm tới một liên kết hướng lên trên: mỗi liên kết hướng lên trên sẽ trỏ đến một khóa duy nhất trong cây mà có các bit làm cho quá trình tìm kiếm hướng tới liên kết đó. Ví dụ, S là khóa duy nhất trong cây mà khớp với mẫu bit 10*11. Do đó nếu khóa ở nút được trỏ tới bởi liên kết hướng trên (mà được chọn tới lần đầu tiên) bằng với khóa tìm kiếm thì quá trình tìm kiếm sẽ kết thúc thành công, ngược lại là không thành công. Với các trie thì tất cả các tìm kiếm kết thúc ở các nút ngoài, khi đó một phép so sánh toàn bộ khóa để xác định quá trình tìm kiếm thành công hay không; phương pháp Patricia tìm kiếm kết thúc tại các liên kết hướng lên phía trên, kế đến một phép so sánh toàn bộ khóa để xác định quá trình tìm kiếm có thành công hay không. Hơn nữa, rất dễ dàng kiểm tra để xác định một liên kết có hướng lên trên hay không, bởi vì các chỉ mục bit trong các nút (do định nghĩa) sẽ giảm khi chúng ta duyệt xuống phía dưới cây. Điều này đưa đến đoạn chương trình sau cho phương pháp Patricia, và đây là một chương trình đơn giản giống như chương trình cho các cây cơ số hay tìm kiếm trên trie.

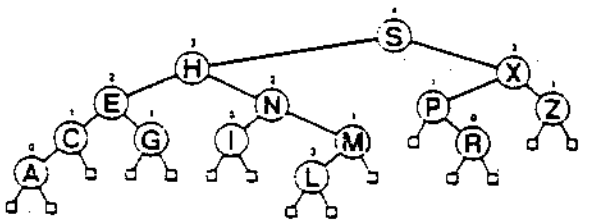
```
type  link=↑node;
      node=record key,info,b:integer; l,r:link end;
var head,z: link;
function patriciasearch(v:integer; x:link):link;
   var p:link;
   begin
   repeat
      p:=x;
      if bits(v,x↑.b,1)=0 then x:=x↑.l else x:=x↑.r;
   until p↑.b<=x;
   patriciasearch:=x;
   end;
```

Hàm này trả về một liên kết tới nút duy nhất mà chứa mẩu tin với khóa v. Chương trình gọi có thể kiểm tra quá trình tìm kiếm có

kết thúc thành công hay không. Do đó để tìm kiếm khóa Z=11010 trong cây nói trên chúng ta rẻ phải và kế đến lên trên ở liên kết phải của X. Khóa ở tại đó không phải là Z vì vậy quá trình tìm kiếm là không thành công.

Hình 17.9 cho thấy kết quả của việc chèn Z=11010 vào cây Patricia của hình 17.8. Như mô tả ở trên, việc tìm kiếm Z kết thúc tại nút chứa X=11000. Do tính chất của cây, X là khóa duy nhất trong cây mà quá trình tìm kiếm kết thúc tại nút đó. Nếu Z được chèn vào thì sẽ có hai nút như thế, vì vậy liên kết hướng lên trên mà đi vào nút chứa X phải được trỏ tới một nút mới chứa Z, và một chỉ mục bit tương ứng tới điểm trái nhất nơi đó X và Z khác nhau với hai liên kết hướng lên trên: một trỏ tới X và liên kết còn lại trỏ tới Z. Điều này y hệt như việc thay thế nút ngoài chứa X bởi một nút trong mới mà X và Z là con của nó trong quá trình chèn vào trie cơ số, và nhánh một hướng được khử bỏ nhờ vào chỉ mục bit.

Việc chèn T=10100 minh họa một trường hợp phức tạp hơn nhiều, bạn hãy xem hình 17.10. Quá trình tìm kiếm kết thúc tại P=10000 cho thấy P là khóa duy nhất trong cây với mẫu bit 10*0*. Bây giờ T và P khác nhau tại bit 2, một vị trí mà bị bỏ qua trong suốt quá trình tìm kiếm. Điều kiện chỉ mục bit giảm khi duyệt xuống phía dưới cây buộc T phải được chèn vào giữa X và P, và có một con trỏ đến T tương ứng với bit 2 của riêng nó, cẩn thận chú ý rằng bit 2 được bỏ qua trước khi chèn T hàm ý rằng P và R có cùng giá trị bit 2.

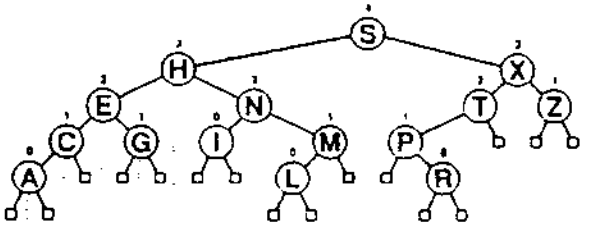


Hình 17.9 *Chèn ngoại vào một cây Patricia*

Những ví dụ trên minh họa chỉ hai trường hợp xảy ra trong quá trình chèn của phương pháp Patricia. Cài đặt sau đây cho thấy các chi tiết:

```
function patriciainsert(v:integer; x:link):link;
   label 0;
   var t,p:link; i:integer;
   begin
   t:=patriciasearch(v,x);
   if v=t↑.key then goto 0;
   i:=maxb;
   while bits(v,i,1)=bits(t↑.key,i,1) do i:=i-1;
   repeat
      p:=x; if bits(v,x↑.b,1)=0 then x:=x↑.l else x:=x↑.r;
   until (x↑.b <= i) or (p↑.b <= x↑.b);
   new(t); t↑.key:=v; t↑.b:=i;
   if bits(v,t↑.b,1)=0
      then begin t↑.l:=t; t↑.r:=x end
      else begin t↑.r:=t; t↑.l:=x end;
   if bits(v,p↑.p,1)=0 then p↑.l:=t else p↑.r:=t;
0: patriciainsert:=t;
   end;
```

(Chương trình này giả sử rằng head được khởi tạo với giá trị của trường khóa là 0, nó có một chỉ mục bit maxb và cả hai liên kết đều trỏ đến chính nó. Trước tiên, chúng ta thực hiện quá trình tìm kiếm để tìm khóa mà phải phân biệt với v. Các điều kiện x↑.b<=i và p↑.b<=x↑.b đặc trưng tình huống tươnng ứng với

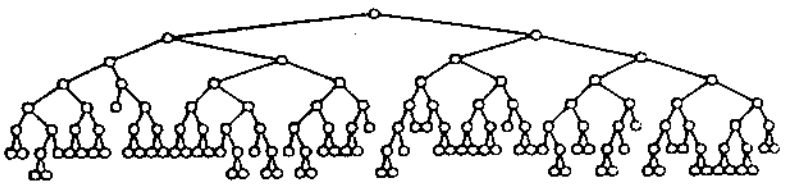


Hình 17.10 *Chèn nội vào một cây Patricia*

hình 17.10 và 17.9. Kế đến chúng ta xác định vị trí bit bên trái nhất mà chúng khác nhau, duyệt hướng xuống dưới cây tới điểm đó và chèn vào một nút mới chứa v ở điểm đó.

Patricia là phương pháp tìm kiếm cơ số tinh tuý nhất: nó dàn xếp để nhận ra các bit khác với khóa tìm kiếm và xây dựng chúng thành một cấu trúc dữ liệu (không có các nút dư thừa), nhờ đó mà tìm ra khóa duy nhất trong cấu trúc dữ liệu mà khóa tìm kiếm có thể bằng khóa đó. Rõ ràng kỹ thuật dùng trong phương pháp Patricia có thể được dùng trong trie tìm kiếm cơ số nhị phân để khử sự phân nhánh một hướng, nhưng điều này chỉ làm trầm trọng thêm vấn đề dạng nút bội (multiple-node-type). Hình 17.11 cho thấy cây Patricia cùng một tập các khóa được dùng để xây dựng trie trong hình 17.6, cây trong hình 17.11 không những chỉ có 44% nút mà còn hoàn toàn cân bằng.

Không giống như phương pháp tìm kiếm trên cây nhị phân chuẩn, các phương pháp cơ số không phụ thuộc vào thứ tự chèn các khóa vào; chúng chỉ phụ thuộc vào cấu trúc của chính các khóa. Với phương pháp Patricia sự thay thế các liên kết hướng lên trên phụ thuộc vào thứ tự chèn, nhưng cấu trúc cây chỉ phụ thuộc vào các bit trong các khóa như trong các phương pháp khác. Do đó ngay cả phương pháp Patricia cũng có sự cố với tập hợp khóa như 001, 0001, 00001, 000001, ... nhưng với các tập khóa chuẩn thì cây sẽ tương đối cân bằng, số lượng kiểm tra bit (ngay cả đối với các khóa rất dài) sẽ xấp xỉ với lgN, trong đó N là số nút trong cây.



Hình 17.11 *Một cây Patricia lớn*

TÍNH CHẤT 17.4 *Một cây Patricia được xây dựng từ N khóa b-bit ngẫu nhiên có N nút và về mặt trung bình đòi hỏi lgN bit phép so sánh cho một quá trình tìm kiếm.*

Như đối với các phương pháp khác trong chương này, sự phân tích cho trường hợp trung bình thì quá khó. Phương pháp Patricia đòi hỏi về mặt trung bình ít hơn trie chuẩn một phép so sánh.

Chức năng hữu dụng nhất của phương pháp tìm kiếm trên trie cơ số là nó có thể hoạt động hiệu quả đối với các khóa có chiều dài biến đổi. Trong cả phương pháp tìm kiếm khác chúng ta đã thấy chiều dài của khóa phải được biết trước, vì vậy thời gian hoạt động phụ thuộc chiều dài cũng như số lượng của các khóa. Khi lựa chọn phương pháp nào thì ta phải dựa vào phương pháp truy xuất bit được dùng. Ví dụ giả sử ta có một máy tính có thể truy xuất hiệu quả các byte dữ liệu 8-bit và chúng ta muốn tìm kiếm trong đám hàng trăm khóa 1000-bit. Khi đó khóa Patricia sẽ đòi hỏi truy xuất 9 hay 10 byte của khóa trong suốt quá trình tìm kiếm, cộng thêm với phép so sánh bằng 125-byte, trong khi phương pháp hash đòi hỏi truy xuất tất cả 125 byte của khóa tìm kiếm để tính hàm hash, cộng thêm một vài phép so sánh bằng, và các phương pháp dựa so sánh thì đòi hỏi nhiều phép so sánh bằng các khóa dài. Sự so sánh này cho thấy phương pháp Patricia (hay tìm kiếm trie cơ số với sự khử đi sự phân nhánh một hướng) là phương pháp nên chọn khi có các khóa rất dài.

BÀI TẬP

1. Vẽ cây tìm kiếm số học có được khi chèn các khóa EASY QUESTION theo thứ tự đó vào một cây được khởi tạo trống.

2. Phát sinh một cây tìm kiếm số học 1000 nút và so sánh độ cao và số nút mỗi tầng của nó với cây tìm kiếm nhị phân chuẩn và cây tìm kiếm đỏ-đen (chương 15) được xây dựng cùng một tập khóa.

3. Hãy tìm một tập hợp 12 khóa mà chúng tạo nên một trie tìm kiếm số học cân bằng yếu.

4. Vẽ trie kiếm cơ số có được khi chèn các khóa E A S Y Q U E S T I O N theo thứ tự đó vào một cây được khởi tạo trống.

5. Một vấn đề xảy ra đối với các trie tìm kiếm số học 26-hướng là một số ký tự trong bảng chữ cái thì lại được sử dụng rất thường xuyên. Hãy đề nghị một phương pháp để giải quyết vấn đề này.

6. Mô tả phương pháp xóa một phần tử khỏi cây tìm kiếm cơ số đa hướng.

7. Vẽ cây Patricia có được khi chèn các khóa E A S Y Q U E S T I O N theo thứ tự đó vào một cây được khởi tạo trống.

8. Tìm một tập hợp 12 khóa mà tạo ra được một cây Patricia cân bằng yếu.

9. Viết một chương trình in ra tất cả các khóa trong cây Patricia mà có t bit khởi đầu giống với một khóa tìm kiếm đã cho.

10. Trong các phương pháp cơ số thì phương pháp nào thích hợp để viết chương trình in ra các khóa theo thứ tự? Phương pháp nào thì không thích hợp?

18
TÌM KIẾM TRÊN BỘ NHỚ NGOÀI

Các thuật toán tìm kiếm thích hợp cho việc truy xuất các phần tử từ các tập tin lớn thì rất quan trọng đối với ứng dụng trong thực tế. Việc tìm kiếm là thao tác cơ bản trên các tập tin dữ liệu lớn, các tập tin nầy chắc chắn bao gồm các tài nguyên có ý nghĩa quan trọng được cài đặt nhiều lần trên máy tính.

Chúng ta sẽ quan tâm nhiều đến các phương pháp tìm kiếm trên các tập tin lớn được lưu trữ trên đĩa bởi vì thao tác tìm kiếm trên đĩa thì thú vị nhất trong thực tế. Với các thiết bị tuần tự chẳng hạn như các băng từ, việc tìm kiếm thoái hóa thành phương pháp chậm tầm thường: để tìm kiếm một phần tử được lưu trên băng từ người ta không thể làm gì tốt hơn ngoài việc lắp băng từ và đọc nó đến khi nào phần tử được tìm thấy. Đặc biệt, các phương pháp mà chúng ta sẽ nghiên cứu có thể tìm thấy một phần tử từ một đĩa chứa hàng tỷ từ mà chỉ cần hai lần truy xuất đĩa.

Cũng giống như các phương pháp sắp xếp ngoài, khía cạnh hệ thống của việc dùng phần cứng Nhập/Xuất (I/O) phức tạp là nhân tố chính yếu trong tính năng của các phương pháp tìm kiếm ngoài (tức tìm kiếm trên bộ nhớ ngoài), nhưng chúng ta không có điều kiện nghiên điều đó một cách chi tiết. Đối với các thuật toán sắp xếp thì các phương pháp ngoài hoàn toàn khác hẳn với các phương pháp nội (sắp xếp trong RAM), nhưng ngược lại chúng ta sẽ thấy các phương pháp tìm kiếm ngoài là các mở rộng logic của các phương pháp tìm kiếm nội mà chúng ta đã nghiên cứu.

Tìm kiếm là một thao tác cơ bản cho các thiết bị đĩa. Các tập tin được quản lý sao cho có thể lợi dụng các đặc trưng thiết bị cụ thể để truy xuất thông tin càng hiệu quả càng tốt. Như đã qui ước

với các thuật toán sắp xếp, chúng sẽ làm việc với một mô hình đĩa đơn giản và không chính xác hoàn toàn như trong thực tế, mục đíc chính của chúng ta là giảng giải các đặc trưng chính của các phương pháp cơ bản. Việc xác định phương pháp tìm kiếm ngoài tốt nhất cho một ứng dụng cụ thể thì vô cùng phức tạp và rất phụ thuộc vào các đặc trưng của phần cứng (và phần mềm hệ thống), vì vậy đó là vấn đề vượt ngoài phạm vi của quyển sách nầy. Tuy nhiên chúng ta cũng đề nghị một vài cách tiếp cận tổng quát để có thể vận dụng trong thực tế.

Đối với nhiều ứng dụng, chúng ta sẽ thường xuyên dùng các thao tác như sửa chữa, thêm vào, xóa bớt hay quan trọng nhất là muốn truy xuất nhanh một mẩu thông tin rất nhỏ bên trong các tập tin vô cùng lớn. Trong chương nầy, chúng ta sẽ kiểm tra một vài phương pháp cho các hoàn cảnh động như thế, các phương pháp nầy cung cấp một số thuận lợi dựa trên các phương pháp không phức tạp mà cây tìm kiếm nhị phân và băm cung cấp trên tìm kiếm nhị phân và tìm kiếm tuần tự.

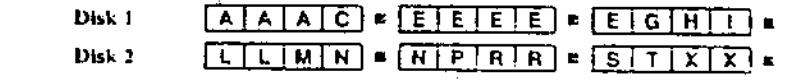
Một sự tập hợp thông tin rất lớn được xử lý bằng cách dùng máy tính sẽ được gọi là một cơ sở dữ liệu (database). Một hướng nghiên cứu lớn là thâm nhập vào các phương pháp xây dựng, bảo trì và sử dụng các cơ sở dữ liệu. Tuy nhiên, các cơ sở dữ liệu lớn có tính ì rất cao: mỗi cơ sở dữ liệu lớn được xây dựng dựa trên một chiến lược truy tìm cụ thể, phải trả một giá rất đắt để xây dựng lại cơ sở dữ liệu mà dùng các chiến lược truy tìm khác. Với lý lẻ nầy, các phương pháp tĩnh trước đây được dùng rộng rãi và người ta cũng thích duy trì chúng, mặc dù các phương pháp động gần đây đang được bắt đầu được dùng cho các cơ sở dữ liệu mới.

Các hệ thống ứng dụng cơ sở dữ liệu thường cung cấp nhiều thao tác phức tạp hơn việc tìm kiếm đơn giản một phần tử dựa vào một khóa đơn. Các thao tác tìm kiếm thường được dựa trên các tiêu chuẩn bao gồm nhiều hơn một khóa và sẽ trả về một số lượng lớn các mẩu tin. Trong các chương sau chúng ta sẽ xem một vài ví dụ về các thuật toán mà thích hợp cho các yêu cầu tìm kiếm thuộc dạng nầy, nhưng các yêu cầu tìm kiếm tổng quát đã đủ phức tạp mặc dù chỉ đơn giản là tìm kiếm tuần tự trên toàn bộ cơ sở dữ liệu, kiểm tra mỗi mẩu tin xem nó có thỏa mãn các điều kiện nào đó hay không.

Các phương pháp mà chúng ta sẽ thảo luận là các phương pháp rất quan trọng về mặc thực hành khi cài đặt các hệ thống tập tin lớn mà trong đó mỗi tập tin có một chỉ danh duy nhất và mục đích của hệ thống tập tin là cung cấp khả năng truy xuất hiệu quả, chèn và xóa dựa trên chỉ danh đó. Mô hình của chúng ta sẽ khảo sát đĩa lưu trữ được chia thành các trang, các khối liên tục của thông tin mà có thể được truy xuất hiệu quả bằng phần cứng đĩa. Mỗi trang sẽ lưu trữ nhiều mẩu tin; nhiệm vụ của chúng ta là quản lý các mẩu tin trong các trang bằng một phương pháp mà bất kỳ một mẩu tin nào cũng có thể được truy xuất bằng cách chỉ truy xuất một vài trang. Chúng ta giả sử rằng thời gian nhập xuất (I/O) cần thiết khi đọc một trang hoàn toàn trội hẳn so với thời gian xử lý cần thiết để thực hiện bất kỳ tính toán liên quan đến trang đó. Như đã lưu ý ở trên, mô hình nầy thì rất đơn giản trên nhiều phương diện, nhưng nó cũng giữ lại đủ các đặc trưng của các thiết bị lưu trữ ngoài thông thường để cho chúng ta khảo sát một vài phương pháp cơ bản thường dùng.

TRUY XUẤT TUẦN TỰ CÓ CHỈ MỤC

Việc tìm kiếm trên đĩa từ một cách tuần tự là mở rộng tự nhiên của các phương pháp tìm kiếm tuần tự cơ bản đã được xem xét trong Chương 14: các mẩu tin được lưu trữ theo thứ tự tăng của khóa, và các thao tác tìm kiếm chỉ đơn giản là lần lượt đọc từng mẩu tin một đến khi gặp một mẩu tin chứa một khóa lớn hơn hay bằng khóa tìm kiếm. Ví dụ, nếu các khóa tìm kiếm của chúng ta là E X T E R N A L S E A R C H I N G E X A M P L E và chúng ta có các đĩa có khả năng lưu trữ ba trang và mỗi trang bốn mẩu tin, thì chúng ta có cấu hình như trong Hình 18.1. (Cũng giống như với các phương pháp sắp xếp ngoài, chúng ta phải xem xét các ví dụ rất nhỏ để hiểu các thuật toán, và suy nghĩ về các ví dụ rất lớn để đánh giá tính năng của chúng.) Hiển nhiên, phương pháp tìm

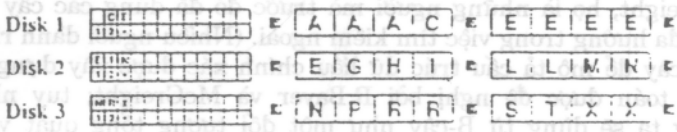


Hình 18.1 *Truy xuất tuần tự*

kiếm tuần tự thuần túy thì không hấp dẫn bởi vì chẳng hạn như khi tìm kiếm W trong Hình 18.1 sẽ đòi hỏi đọc toàn bộ tất cả các trang.

Để cải tiến mạnh tốc độ của một thao tác tìm kiếm, chúng ta có thể duy trì cho mỗi đĩa một "chỉ mục" cho biết các khóa thuộc về trang nào trên đĩa đó, như trong Hình 18.2. Trang đầu của mỗi đĩa là chỉ mục của nó: các chữ nhỏ cho thấy chỉ có giá trị khóa được lưu trữ chứ không phải toàn bộ mẩu tin và các số nhỏ là các chỉ mục trang (0 nghĩa là trang đầu trên đĩa, 1 là trang kế tiếp, ...). Trong chỉ mục, mỗi số trang xuất hiện phía dưới giá trị của khóa cuối cùng trên trang trước đó. (Khoảng trắng là một khóa cầm canh nhỏ hơn tất cả các khóa còn lại, và ký hiệu '+' nghĩa 'nhìn đến đĩa kế tiếp'.) Do đó, chẳng hạn chỉ mục cho đĩa 2 cho thấy rằng trang đầu tiên của nó chứa các mẩu tin bao gồm các khóa nằm ở giữa E và I, và trang thứ hai của nó chứa các mẩu tin bao gồm các mẩu tin gồm các khóa nằm giữa I và N. Có thể nhiều khóa và các chỉ mục trang trên một trang chỉ mục hơn các mẩu tin trên một trang "dữ liệu"; thật ra, chỉ mục cho toàn bộ đĩa chỉ nên đòi hỏi một vài trang.

Để giải quyết tốt hơn vấn đề tìm kiếm, những chỉ mục nầy có lẻ phải đi đôi với một "chỉ mục chính" (master index) để cho biết khóa nằm ở trên đĩa nào. Với ví dụ của chúng ta, chỉ mục chính nói rằng đĩa 1 chứa các khóa nhỏ hơn hay bằng E, đĩa 2 chứa các khóa nhỏ hơn hay bằng N (nhưng không nhỏ hơn E), và đĩa 3 chứa các khóa nhỏ hơn hay bằng X (nhưng không nhỏ hơn N). Chỉ mục chính thì thường đủ nhỏ sao cho có thể lưu nó trong bộ nhớ, sao cho hầu hết các mẩu tin có thể được tìm thấy bằng cách truy xuất chỉ hai trang, trong đó một trang là chỉ mục trên đĩa thích hợp, và một trang chứa mẩu tin thích hợp. Ví dụ, khi tìm khóa W thì trước tiên là đọc chỉ mục trang từ đĩa 3, kế đến là đọc trang trang dữ liệu thứ hai từ đĩa 3 và chính là đĩa duy nhất mà có thể chứa W. Việc tìm kiếm các khóa xuất hiện trong chỉ mục đòi hỏi đọc ba trang: trang chỉ mục cộng với hai trang kề bên giá trị khóa trong chỉ mục. Nếu không có các khóa trùng trong một tập tin thì có thể tránh sự truy xuất trang phụ trợ. Nói cách khác, nếu có nhiều khóa bằng nhau trong một tập tin thì sẽ có khả năng xảy ra nhiều



Hình 18.2 *Truy xuất tuần tự có chỉ mục*

lần truy xuất trang (các mẩu tin mà có các khóa bằng nhau có khả năng trải ra nhiều trang).

Bởi vì có sự phối hợp sự tổ chức khóa tuần tự với các truy xuất có chỉ mục, phương pháp nầy được gọi là truy xuất tuần tự có chỉ mục. Đây là phương pháp được chọn cho các ứng dụng mà trong đó sự thay đổi thông tin trong cơ sở dữ liệu có khả năng xảy ra thường xuyên. Bất lợi của việc dùng phương pháp nầy là nó không linh động. Ví dụ khi thêm B vào cấu hình như trên thì toàn bộ cơ sở dữ liệu phải được xây dựng lại, để có các vị trí mới cho nhiều khóa và các giá trị mới cho các các chỉ mục.

TÍNH CHẤT 18.1 *Một thao tác tìm kiếm trên một tập tin tuần tự có chỉ mục đòi hỏi số lần truy xuất đĩa là một hằng số, nhưng một thao tác chèn có thể bao gồm việc bố trí lại toàn bộ tập tin.*

Như thường lệ, "hằng số" được đề cập ở đây phụ thuộc vào số đĩa, kích thước tương đối của các mẩu tin, các chỉ mục và các trang. Ví dụ, một tập tin lớn bao gồm các khóa một từ chắc chắn không thể được lưu trữ chỉ trong một đĩa bằng phương pháp như thế để thừa nhận số lần truy xuất đĩa là một hằng số. Hay là có thể lấy một ví dụ rất buồn cười, ở một thái cực khác, một số lượng lớn đĩa rất nhỏ, mỗi đĩa có thể lưu trữ chỉ một mẩu tin rất khó truy xuất nó.

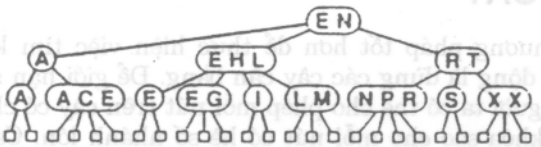
CÁC B-CÂY

Một phương pháp tốt hơn để thực hiện việc tìm kiếm trong hoàn cảnh động là dùng các cây cân bằng. Để giới hạn số lần truy xuất đĩa, người ta có thể cho phép mỗi nút trên cây có chứa một số lượng lớn khóa sao cho mỗi nút có hệ số nhánh lớn. Các cây như thế được đặt tên là B-cây, B-cây được khám phá bởi R.Bayer và

McCreight, họ là những người mà trước đó đã dùng các cây cân bằng đa hướng trong việc tìm kiếm ngoài. (Nhiều người dành riêng từ B-cây để mô tả cấu trúc dữ liệu chính xác được xây dựng bởi thuật toán được đề nghị bởi R.Bayer và McCreight; tuy nhiên chúng ta sẽ dùng từ B-cây như một đối tượng tổng quát với ý nghĩa "các cây cân bằng ngoài".)

Thuật toán từ trên xuống mà chúng ta đã dùng cho các cây 2-3-4 (xem chương 15) có thể mở rộng cho trường hợp có nhiều khóa mỗi nút: giả sử rằng có từ 1 đến M-1 khóa mỗi nút (và như vậy sẽ có từ 2 đến M liên kết mỗi nút). Qui trình tìm kiếm thì tương tự như đối với các cây 2-3-4: để di chuyển từ một nút tới nút kế tiếp, trước tiên tìm đoạn riêng cho khóa tìm kiếm trong nút hiện hành và kế đến chuyển tới liên kết tương ứng để đạt được nút kế tiếp. Tiếp tục quá trình nầy đến khi chạm tới một nút ngoài, kế đó chèn khóa mới vào nút trong cuối cùng vừa được chạm. Cũng như với các cây 2-3-4 từ trên xuống, cần phải "tách" các nút mà bị "đầy" trên con đường duyệt cây hướng xuống dưới: bất kỳ lúc nào gặp một k-nút tiếp xúc với một M-nút, chúng ta sẽ thay nó bằng một (k+1)-nút tiếp xúc với hai (M/2)-nút (để cho dễ tách chúng ta giả sử M chẵn). Quá trình trên bảo đảm rằng khi chạm tới đáy thì sẽ có chổ để chèn thêm nút mới vào.

Hình 18.3 minh họa một B-cây được xây dựng từ tập các khóa mẫu của chúng ta khi M=4. Cây nầy có 13 nút, mỗi nút tương ứng với một trang đĩa. Mỗi nút phải chứa các liên kết cũng như các mẩu tin. Mặc dù khi chọn M=4 thì ta có trường hợp cây 2-3-4 quen thuộc, nhưng điều nầy cũng nhấn mạnh rằng: chúng ta không thể bố trí bốn mẩu tin cho mỗi trang mà chỉ bố trí ba mẩu tin để dành chổ cho các liên kết. Trữ lượng không gian lưu trữ thường dùng phụ thuộc vào kích thước tương đối của các mẩu tin và các liên kết. Dưới đây chúng ta sẽ xem một phương pháp có thể tránh sự xáo trộn nầy giữa các mẩu tin và các liên kết.



Hình 18.3 *Một B-cây*

Cũng như chúng ta đã lưu chỉ mục chính trong bộ nhớ đối với tìm kiếm tuần tự có chỉ mục, bây giờ chúng ta hoàn toàn có lý khi lưu nút gốc của B-cây trong bộ nhớ chính. Đối với B-cây trong hình 18.3, điều này khẳng định rằng gốc của cây con chứa các mẩu tin có khóa nhỏ hơn hay bằng E trong trang 0 của đĩa 1, gốc của cây con với các khóa nhỏ hơn hay bằng N (nhưng không nhỏ hơn E) trong trang 1 của đĩa 1, và gốc của cây con có các khóa lớn hơn hay bằng N trong trang 2 của đĩa 1. Các nút khác trong ví dụ của chúng ta có thể được lưu trữ như trong hình 18.4.

Các nút được gán tới các trang của đĩa trong ví dụ này bởi xử lý đơn giản là duyệt xuống dưới cây, làm việc từ phải tới trái ở mỗi tầng, gán các nút tới đĩa 1, kế đến là đĩa 2, ... Chúng ta tránh lưu trữ các liên kết NULL bằng cách theo dõi khi nào thì chạm tới tầng đáy: trong trường hợp này, tất cả các nút trên các đĩa 2, 3, và 4 đều chứa các liên kết NULL (mà không cần lưu trữ). Trong một ứng dụng cụ thể có thể giải quyết vấn đề này khác hơn. Ví dụ, một cách tốt hơn để tránh tìm kiếm xuyên qua đĩa 1 bằng cách trước tiên gán trang 0 tới tất cả các đĩa. Thật ra, nhiều chiến lược tinh vi hơn được cần đến bởi vì tính chất động của việc xây dựng cây (hãy xem sự khó khăn của việc cài đặt một thủ tục split mà đáp ứng một trong các chiến lược trên).

TÍNH CHẤT 18.2 *Một thao tác tìm kiếm hay chèn vào một B-cây cấp M có N mẩu tin đòi hỏi ít hơn* $\log_{M/2} N$ *lần truy xuất đĩa, xét về phương diện thực hành thì số này nhỏ hơn một hằng số (miễn là M không nhỏ).*

Tính chất này được suy ra từ quan sát rằng tất cả các nút "trong" của B-cây (các nút không phải gốc và cũng không phải lá) có chứa từ M/2 tới M khóa, bởi vì chúng được tạo từ việc tách một

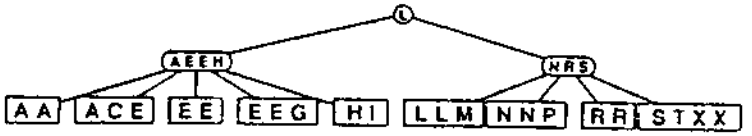
Hình 18.4 *Truy xuất một B-cây*

nút đầy có M khóa, và chỉ có thể tăng kích thước (khi một nút thấp hơn được tách). Trong trường hợp xấu nhất, các nút này tạo nên một cây đầy đủ cấp M/2, và trường hợp này thì số lần truy xuất đĩa xấp xỉ với $\log_{M/2}N$.

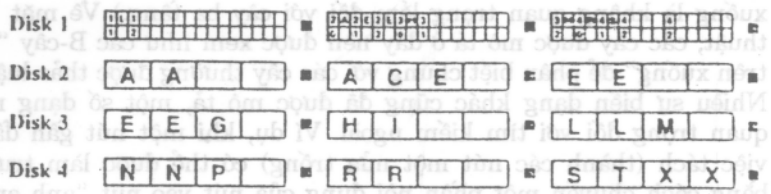
TÍNH CHẤT 18.3 *Một B-cây cấp M được xây dựng từ N mẩu tin ngẫu nhiên có lẻ có khoảng 1.44N/M nút.*

Việc chứng minh tính chất này vượt ngoài phạm vi của quyển sách này, nhưng chú ý rằng kích thước không gian lưu trữ bị phí đi là khoảng N, trong trường hợp xấu nhất khi tất cả các nút đầy khoảng một nửa.

Trong ví dụ trên chúng ta bị bắt buộc chọn M=4 bởi vì cần giữ chỗ cho các liên kết của các nút. Nhưng chúng ta kết luận không dùng các liên kết trong hầu hết các nút, bởi vì hầu hết các nút trong trong B-cây là các nút ngoài và hầu hết các liên kết đều là NULL. Hơn nữa một giá trị M lớn hơn nhiều có thể được dùng ở tầng cao hơn của cây nếu chúng ta chỉ lưu các khóa (chứa không phải các mẩu tin đầy đủ) trong các nút trong giống như trong truy xuất tuần tự có chỉ mục. Trong ví dụ của chúng ta, để thấy lợi ích của nhận xét này, giả sử rằng có thể trang bị bảy khóa và tám liên kết trên một trang, khi đó ta có thể dùng M=8 cho các nút trong và M=5 cho các nút ở tầng đáy (không phải M=4 bởi vì không có chỗ cho các các liên kết cần được giành riêng ở đáy). Một nút ở đáy được tách khi một mẩu tin thứ năm được thêm vào nó (tách thành một nút hai mẩu tin và một nút ba mẩu tin); quá trình tách kết thúc khi "chèn" khóa của mẩu tin giữa vào trong nút nói trên, ở đó có chỗ bởi vì cây đã hoạt động như một B-cây chuẩn với M=8 (số khóa được lưu trữ, không phải số mẩu tin). Quá trình này đưa đến cây như trong hình 18.5.



Hình 18.5 *Một B-cây chỉ có các mẩu tin ở các nút ngoài*



Hình 18.6 *Truy xuất một B-cây chỉ có các mẩu tin ở các nút ngoài*

Cũng với dạng tổ chức cấu trúc này, "bộ phận chỉ mục" (chứa các khóa và các liên kết) có thể tách biệt khỏi các mẩu tin như trong tìm kiếm tuần tự có chỉ mục. Hình 18.6 cho thấy làm thế nào để lưu trữ cây trong hình 18.5: nút gốc được lưu trên trang 0 của đĩa 1 (có chỗ cho nó bởi vì cây trong hình 18.5 có ít hơn cây trong hình 18.3 một nút), tuy nhiên trong hầu hết các ứng dụng nó có khả năng được lưu trong bộ nhớ chính.

Bây giờ chúng ta có hai giá trị của M, một cho các nút trong mà xác định hệ số nhánh của cây (M_I) và một cho các nút ở tầng đáy mà xác định sự phân phối các mẩu tin tới các trang (M_B). Để tối thiểu hóa số lần truy xuất đĩa, chúng ta muốn cho cả M_I lẫn M_B càng lớn càng tốt. Mặt khác, chúng ta không muốn làm M_I quá lớn, bởi vì khi đó hầu hết các nút của cây sẽ trống nhiều và chỗ lưu trữ sẽ phí, chúng ta cũng không muốn M_B quá lớn, bởi vì điều này sẽ hạn chế quá trình tìm kiếm tuần tự của các nút ở tầng đáy. Thông thường thì tốt nhất là liên hệ cả M_I lẫn M_B tới kích thước trang. Sự lựa chọn hiển nhiên cho M_B là số mẩu tin mà có thể lắp đầy một trang (cộng một thêm): mục đích của sự tìm kiếm là tìm thấy trang chứa mẩu tin muốn tìm. Nếu M_I được chọn là số khóa mà có thể lắp đầy hai hay bốn trang, khi đó B-cây gần như chỉ sâu ba tầng, ngay cả đối với các tập tin rất lớn (một cây ba tầng với M_I=2048 có thể chứa 1024^3 hay trên một tỷ đầu vào). Nhưng nên nhớ rằng nút gốc của cây được truy xuất thường xuyên và nó được lưu trong bộ nhớ, vì vậy chỉ cần hai lần truy xuất đĩa là có thể tìm thấy bất kỳ phần tử nào trong tập tin.

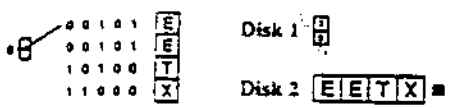
Như đã lưu ý trong phần cuối của chương 15, một phương pháp chèn "từ dưới lên trên" phức tạp hơn thường được dùng cho các B-cây (mặc dù sự khác biệt giữa các phương pháp từ trên

xuống là không quan trọng lắm đối với cây ba tầng.) Về mặt kỹ thuật, các cây được mô tả ở đây nên được xem như các B-cây "từ trên xuống" để phân biệt chúng với các cây thường được thảo luận. Nhiều sự biến dạng khác cũng đã được mô tả, một số dạng rất quan trọng đối với tìm kiếm ngoài. Ví dụ, khi một nút gần đầy, việc tách (thành các nút một nửa trống) có thể được làm trước bằng cách chuyển một phần nội dung của nút vào nút "anh em" của nó (nếu nút này không quá đầy). Thao tác này dẫn đến sự tận dụng không gian lưu trữ bên trong nút tốt hơn, và đó chính là mối quan tâm chính trong các ứng dụng tìm kiếm trên một đĩa trữ lượng lớn.

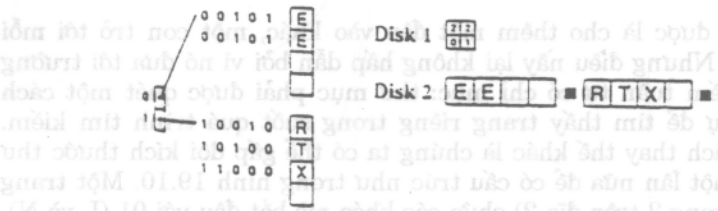
BĂM MỞ RỘNG

Một mở rộng của các thuật toán tìm kiếm số học có thể thay thế cho các B-cây khi ứng dụng vào các phương pháp tìm kiếm ngoài đã được phát triển trong năm 1978 bởi R.Fagin, J.Nievergelt, N.Pippenger, và R.Strong. Phương pháp này được gọi là **Băm mở rộng**, nó đòi hỏi hai lần truy xuất đĩa cho mỗi thao tác tìm kiếm trong các ứng dụng thông thường. Như đối với các B-cây, các mẩu tin của chúng ta được lưu trữ trong các trang mà được tách thành hai phần khi chúng đầy; cũng giống như với phương pháp truy xuất tuần tự có chỉ mục, chúng ta duy trì một chỉ mục để có thể truy xuất tới các trang chứa các mẩu tin cần tìm kiếm. Phương pháp băm mở rộng tổ hợp các phương pháp này bằng cách dùng các tính chất số của các khóa tìm kiếm.

Để hiểu phương pháp băm mở rộng làm việc như thế nào, chúng ta khảo sát quá trình chèn liên tiếp các khóa E X T E R N A L S E A R C H I N G E X A M P L E, và dùng các trang có khả năng chứa bốn mẩu tin. Chúng ta khởi đầu với một "chỉ mục" chỉ



Hình 18.7 *Phương pháp Băm mở rộng: trang đầu tiên*



Hình 18.8 *Phương pháp Băm mở rộng: tách thư mục*

có một đầu vào, tức là một con trỏ tới trang mà sẽ lưu giữ các mẩu tin. Bốn mẩu tin đầu tiên lấp đầy trên một trang, duy trì cấu trúc thông thường như trong hình 18.7.

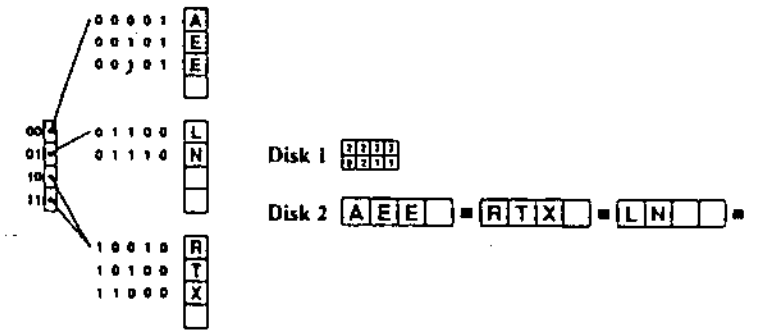
Thư mục trên đĩa 1 cho thấy rằng tất cả các mẩu tin đều ở trên trang 0 của đĩa 2, ở đó chúng được lưu trữ theo thứ tự khóa tăng. Để tham khảo, chúng ta cũng cho các giá trị nhị phân của khóa, dùng sự mã hóa chuẩn của chúng ta, nghĩa là biểu diễn nhị phân năm bit của i cho ký tự thứ i trong bảng chữ cái. Bây giờ thì trang bị đầy và phải được tách theo thứ tự để thêm vào khóa R=10010. Chiến lược thì rất đơn giản: hãy đặt các mẩu tin với các khóa bắt đầu với 0 vào một trang và các khóa bắt đầu với 1 vào trang còn lại. Thao tác nầy bắt buộc phải nhân đôi kích thước của thư mục và di chuyển một nửa các khóa từ trang 0 của đĩa 2 vào một trang mới, bây giờ cấu trúc lưu trữ giống như hình 18.8.

Bây giờ N=01110 và A=00001 có thể được thêm vào, nhưng điều nầy một lần nữa làm đầy trang đầu tiên như trong hình 18.9. Chúng ta lại cần một thao tác tách trước khi chèn L=01100 vào. Để chèn L vào, chúng ta dùng cùng một phương pháp như lần tách đầu tiên, tách trang đầu tiên thành hai phần, một phần cho các khóa bắt đầu với 00 và một phần cho các khóa bắt đầu với 01. Đối với các thư mục thì không rõ phải làm thao tác gì. Một sự thay thế

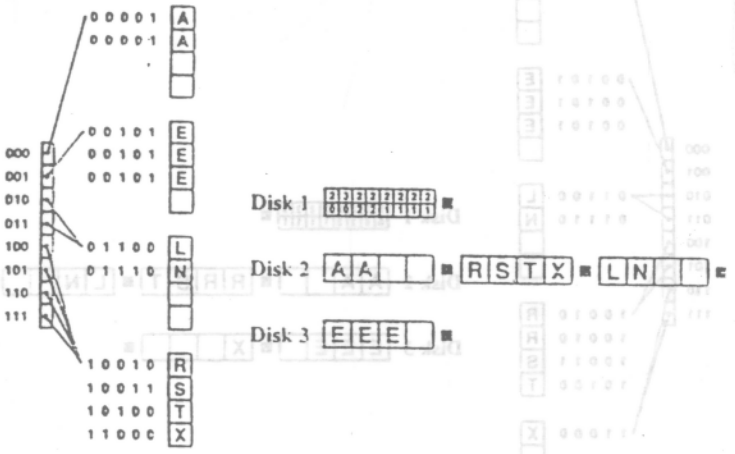
Hình 18.9 *Phương pháp Băm mở rộng: trang đầu tiên đầy một lần nữa*

có thể được là cho thêm một đầu vào khác, một con trỏ tới mỗi trang. Nhưng điều nầy lại không hấp dẫn bởi vì nó đưa tới trường tìm kiếm tuần tự có chỉ mục: thư mục phải được quét một cách tuần tự để tìm thấy trang riêng trong suốt quá trình tìm kiếm. Một cách thay thế khác là chúng ta có thể gấp đôi kích thước thư mục một lần nữa để có cấu trúc như trong hình 19.10. Một trang mới (trang 2 trên đĩa 2) chứa các khóa mà bắt đầu với 01 (L và N), bây giờ trang được tách (trang 0 trên đĩa 2) chứa các khóa bắt đầu với 00 (A, E, và E), và trang chứa các khóa bắt đầu với 1 (R, T, và X) thì không bị ảnh hưởng, mặc dù bây giờ có hai con trỏ tới nó, một con trỏ cho các khóa bắt đầu với 10, và một con trỏ cho các khóa bắt đầu với 11. Bây giờ chúng ta có thể truy xuất bất kỳ mẩu tin nào nhờ vào hai bit đầu tiên của khóa để truy xuất trực tiếp tới thư mục chứa địa chỉ của trang chứa mẩu tin.

Việc duy trì các mẩu tin theo thứ tự trong các trang thì dường như là một bắt buộc quá ngặt, nhưng hãy nhớ lại rằng giả thiết cơ sở của chúng ta là khảo sát nhập/xuất (I/O) đĩa với các đơn vị trang và thời gian xử lý đó thì không đáng kể so với thời gian thời gian nhập hay xuất một trang. Do đó việc duy trì các mẩu tin theo thứ khóa của chúng thì không là một phí tổn thật sự: để thêm một mẩu tin vào một trang, chúng ta phải đọc trang vào bộ nhớ, sửa đổi nó, và ghi trở lại. Thời gian cần thiết để duy trì thứ tự sắp xếp thì không đáng chú ý trong trường hợp thông thường khi các trang không lớn lắm.



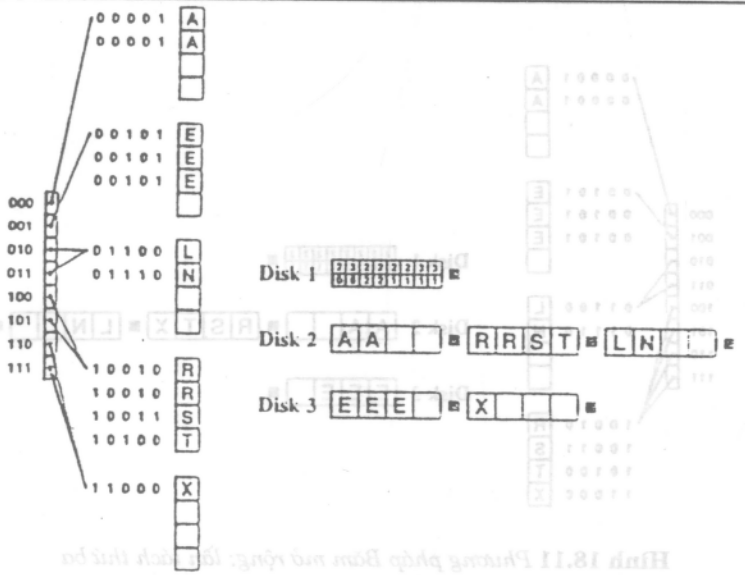
Hình 18.10 *Phương pháp Băm mở rộng: lần tách thứ hai*



Hình 18.11 *Phương pháp Băm mở rộng: lần tách thứ ba*

Tiếp tục xa hơn một tý nữa, chúng ta có thể thêm S=10011 và E=00101 trước khi phải tách một lần nữa để thêm A=00001 vào. Thao tác tách nầy cũng đòi hỏi nhân đôi thư mục để được một cấu trúc như trong hình 18.11. Xử lý nhân đôi thì đơn giản: chỉ cần đọc thư mục cũ, kế đến tạo ra một thư mục mới bằng cách ghi ra hai lần cho mỗi đầu vào của thư mục cũ. Quá trình nầy tạo ra khoảng trống để lưu các con trỏ cho các trang mới vừa được tạo bởi thao tác tách.

Nói chung, cấu trúc được xây dựng bởi phương pháp băm mở rộng bao gồm một **thư mục chứa** 2^d từ (một từ tương ứng với mẫu d-bit) và một tập hợp các **trang lá** mà chứa tất cả các mẩu tin với các khóa bắt đầu với một dãy bit cụ thể (nhỏ hơn hay bằng d bit). Một thao tác tìm kiếm đưa đến việc dùng d bit dẫn đầu của khóa vào chỉ mục của thư mục mà chứa các con trỏ tới các trang lá. Kế đến trang được truy xuất và tìm kiếm (dùng bất kỳ một chiến lược nào) cho mẩu tin. Một trang lá có thể được trỏ tới bởi nhiều hơn một đầu vào của thư mục: chính xác hơn, nếu trang lá chứa tất cả các mẩu tin với khóa bắt đầu bằng k bit cho trước thì sẽ có 2^{d-k} đầu vào trong thư mục trỏ tới nó. Trong hình 18.11

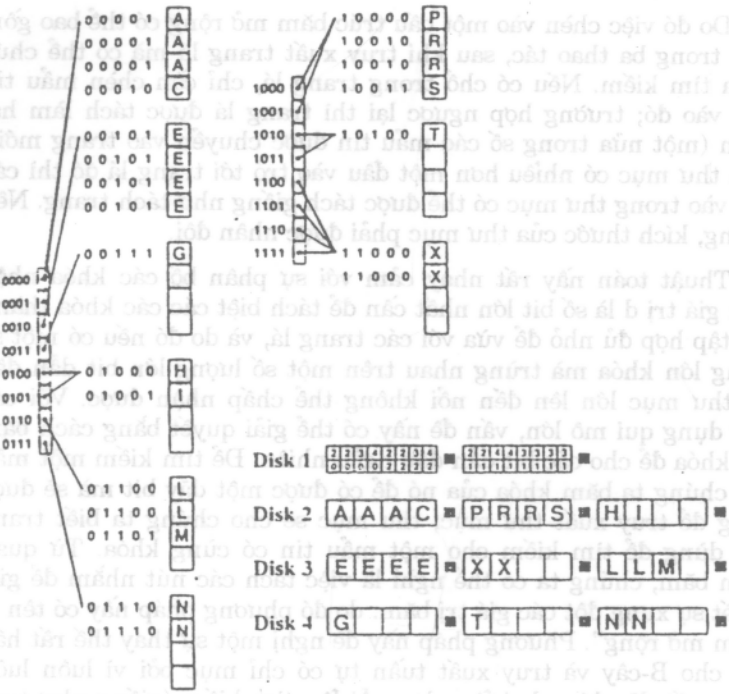


Hình 18.12 *Phương pháp Băm mở rộng: lần tách thứ tư*

chúng ta có d=3, và trang 1 của đĩa 2 chứa tất cả các mẩu tin với các khóa bắt đầu với bit 1, vì vậy có bốn đầu vào của thư mục trỏ tới nó.

Cho tới bây giờ, đối với ví dụ của chúng ta thì mỗi trang một thao tác tách thư mục, nhưng trường hợp tổng quát thì có thể hiếm hơn. Đây chính là bản chất của thuật toán này: các con trỏ trợ giúp trong thư mục cho phép các cấu trúc thao tác động một cách tuyệt diệu. Ví dụ khi chèn R vào cấu trúc trong hình 18.11, trang 1 của đĩa 2 phải bị tách để đáp ứng với năm khóa bắt đầu với 1, nhưng thư mục không cần phát triển, xem hình 8.12. Thay đổi duy nhất đối với thư mục là hai con trỏ cuối cùng được thay đổi để trỏ tới trang 1 trên đĩa, trang mới được tạo để chứa tất cả các khóa trong cấu trúc dữ liệu mà bắt đầu với 11 (khóa X).

Thư mục chỉ chứa hai con trỏ tới các trang, kích thước lưu trữ nhỏ hơn so với các khóa hay các mẩu tin, vì vậy số đầu vào của thư mục lấp đầy một trang sẽ nhiều hơn. Trong ví dụ của chúng ta, chúng ta sẽ giả sử rằng số đầu vào của thư mục chứa trong một trang thì gấp đôi số mẩu tin trong một trang, mặc dù trong thực



Hình 18.13 *Truy xuất của cấu trúc Băm mở rộng*

tế thì tỉ số nầy lớn hơn nhiều. Khi thư mục trãi nhiều hơn một trang, chúng ta dùng cùng một sơ đồ chỉ mục để lưu một "nút gốc" trong bộ nhớ nhằm cho biết vị trí các trang của thư mục, ví dụ nếu thư mục trãi ra trong hai trang thì nút gốc cho biết thư mục của các mẩu tin mà khóa bắt đầu với 0 thì trong trang 0 của đĩa 1, thư mục cho của các mẩu tin bắt đầu với 1 thì trong trang 1 của đĩa 1. Với ví dụ của chúng ta thì sẽ có thao tác tách sau khi chèn vào các khóa C, H, I, N, G, và E, tiếp tục chèn X, A, M, P, và L ta có được cấu trúc đĩa giống như trong hình 18.13. (Chúng ta dành riêng đĩa 1 cho thư mục, mặc dù trong thực tế nó có thể cùng đĩa với các trang khác, trang 0 của đĩa có thể được dành riêng, hoặc là dùng một chiến lược khác.)

Do đó việc chèn vào một cấu trúc băm mở rộng có thể bao gồm một trong ba thao tác, sau khi truy xuất trang lá mà có thể chứa khóa tìm kiếm. Nếu có chỗ trong trang lá, chỉ cần chèn mẩu tin mới vào đó; trường hợp ngược lại thì trang lá được tách làm hai phần (một nửa trong số các mẩu tin được chuyển vào trang mới). Nếu thư mục có nhiều hơn một đầu vào trỏ tới trang lá đó thì các đầu vào trong thư mục có thể được tách giống như tách trang. Nếu không, kích thước của thư mục phải được nhân đôi.

Thuật toán nầy rất nhạy cảm với sự phân bố các khóa nhập xấu: giá trị d là số bit lớn nhất cần để tách biệt các các khóa thành các tập hợp đủ nhỏ để vừa với các trang lá, và do đó nếu có một số lượng lớn khóa mà trùng nhau trên một số lượng lớn bit dẫn đầu thì thư mục lớn lên đến nỗi không thể chấp nhận được. Với các ứng dụng qui mô lớn, vấn đề nầy có thể giải quyết bằng cách băm các khóa để cho các bit dẫn đầu ngẫu nhiên. Để tìm kiếm một mẩu tin, chúng ta băm khóa của nó để có được một dãy bit mà sẽ được dùng để truy xuất thư mục; thư mục sẽ cho chúng ta biết trang nào dùng để tìm kiếm cho một mẩu tin có cùng khóa. Từ quan điểm băm, chúng ta có thể nghĩ là việc tách các nút nhằm để giải quyết sự xung đột các giá trị băm: do đó phương pháp nầy có tên là "băm mở rộng". Phương pháp nầy đề nghị một sự thay thế rất hấp dẫn cho B-cây và truy xuất tuần tự có chỉ mục bởi vì luôn luôn truy xuất đĩa đúng hai lần cho mỗi lần tìm kiếm (giống như truy xuất tuần tự có chỉ mục) trong khi vẫn duy trì được khả năng chèn hiệu quả (giống như B-cây) mà không phí nhiều chỗ lưu trữ.

TÍNH CHẤT 18.4 *Với các trang mà có thể lưu trữ M mẩu tin, phương pháp băm mở rộng đòi hỏi khoảng 1.44(N/M) trang cho một tập tin với N mẩu tin. Thư mục có lẻ yêu cầu*

$$\frac{N^{1+\frac{1}{M}}}{M}$$ *đầu vào.*

Sự phân tích cho tính chất nầy là mở rộng phức tạp của sự phân tích về trie trong chương trước. Khi M lớn thì sẽ phí nhiều chỗ lưu trữ giống như các B-cây, nhưng với M nhỏ thì thư mục sẽ trở nên rất lớn.

Ngay cả với phương pháp băm, các bước trợ giúp phải thêm vào nếu có một số lượng lớn khóa trùng nhau. Chúng nó có thể làm thư mục lớn lên một cách thông minh; và thuật toán tách hoạt động nếu các khóa bằng nhau không thể chứa đủ vào một trang. (Điều nầy cũng đã xuất hiện trong trong ví dụ của chúng ta, bởi vì chúng ta có năm khóa E trong ví dụ.) Nếu có nhiều khóa bằng nhau thì chúng ta có thể giả sử các khóa phân biệt trong cấu trúc dữ liệu và đặt thêm vào các con trỏ tới các danh sách liên kết của các mẩu tin có khóa bằng nhau trong các trang lá. Để thấy rõ hơn, bạn hãy xem điều gì xảy ra nếu chèn E vào cấu trúc trong hình 18.13.

Một tình huống nữa có thể xảy ra là khi chèn vào một khóa mới thì sẽ làm cho thư mục bị tách nhiều hơn một lần. Điều nầy xảy ra khi không đủ bit để phân biệt các khóa trong một trang đầy. Ví dụ như nếu chèn hai khóa với giá trị D=00100 vào cấu trúc băm mở rộng trong hình 18.12 thì sẽ xảy ra hai lần tách thư mục bởi vì cần năm bit để phân biệt D và E (bit thứ tư thì không giúp đỡ được gì). Đây là một vấn đề đơn giản chỉ liên quan đến việc cài đặt nhưng phải không được phép quên nó.

BỘ NHỚ ẢO

Phương pháp "dễ dàng" đã được thảo luận ở cuối chương 13 cho việc sắp xếp ngoài có thể được áp dụng trực tiếp và dễ dàng tới vấn đề tìm kiếm. Một bộ nhớ ảo thì không có gì khác hơn là một phương pháp tìm kiếm ngoài nói chung: cho một địa chỉ, hãy trả về thông tin kết hợp với địa chỉ đó. Tuy nhiên việc dùng trực tiếp của bộ nhớ ảo thì không được giới thiệu như một áp dụng tìm kiếm dễ. Như đã chú ý trong chương 13, các bộ nhớ ảo thực hiện tốt nhất khi hầu hết các truy xuất đều tương đối gần với các truy xuất trước đó. Các thuật toán sắp xếp có thể được sửa lại cho thích hợp, nhưng tính chất tự nhiên của các phương pháp tìm kiếm thì phù hợp hơn đối với các cơ sở dữ liệu.

BÀI TẬP

1. Hãy cho biết nội dung của B-cây có được khi chèn các khóa E A S Y Q U E S T I O N theo thứ tự đó vào một cây khởi động trống với M=5.

2. Hãy cho biết nội dung của B-cây có được khi chèn các khóa E A S Y Q U E S T I O N theo thứ tự đó vào một cây khởi động trống với M=6. Dùng phương pháp mà tất cả các mẩu tin đều được lưu trong các nút ngoài.

3. Vẽ một B-cây có được khi chèn mười sáu khóa trùng nhau được chèn vào một cây khởi tạo trống với M=5.

4. Giả sử rằng một trang của cơ sở dữ liệu bị phá hủy. Hãy mô tả cách giải quyết của bạn đối với sự cố này cho mỗi cấu trúc B-cây trong chương này.

5. Hãy cho biết nội dung của bảng băm mở rộng có được khi chèn các khóa E A S Y Q U E S T I O N theo thứ tự đó vào một bảng được khởi tạo trống, bốn mẩu tin mỗi trang. (Theo ví dụ trong chương, không băm nhưng dùng biểu diễn nhị phân 5 bit của i cho ký tự thứ i.)

6. Hãy đưa ra một dãy khóa phân biệt mà tạo nên một thư mục băm mở rộng có được từ một bảng khởi tạo trống tới kích thước 16, mỗi trang của bảng chứa được ba mẩu tin.

7. Phác họa phương pháp xóa một phần tử từ một bảng băm mở rộng.

8. Tại sao các B-cây "từ trên xuống" tốt hơn các B-cây "từ dưới lên" đối với việc truy xuất đồng hành tới dữ liệu? (ví dụ, giả sử hai chương trình chèn cùng một nút mới vào cùng một thời điểm.)

9. Cài đặt các thủ tục search (tìm kiếm) và insert (chèn) để tìm kiếm nội bằng cách dùng phương pháp băm mở rộng.

10. So sánh chương trình của bài tập trước với phương pháp tìm kiếm băm kép và trie cơ số đối với các ứng dụng tìm kiếm nội.

19
TÌM KIẾM CHUỖI

Dữ liệu được xử lý thường không phân rã một cách logic thành các mẩu tin độc lập với các phần nhỏ có thể phân biệt được với nhau. Kiểu dữ liệu này có thể được đặc trưng duy nhất như một chuỗi (string) : đó là một dãy ký tự tuyến tính (có thể là rất dài).

Rõ ràng chuỗi là thành phần trung tâm trong các hệ xử lý văn bản, là các hệ cung cấp nhiều khả năng để thao tác văn bản. Các hệ như vậy xử lý các chuỗi văn bản, các chuỗi này được định nghĩa một cách "lỏng lẻo" như là các dãy chữ, số và ký tự đặc biệt. Những đối tượng này có thể rất lớn (ví dụ, cuốn sách này chứa trên một triệu ký tự), và các thuật toán hiệu quả sẽ đóng một vai trò quan trọng trong việc thao tác trên các chuỗi đó.

Một kiểu chuỗi khác là chuỗi nhị phân, đó là một dãy các giá trị 0 và 1. Trong một ngữ cảnh nào đó, nó đơn giản chỉ là một kiểu chuỗi văn bản đặc biệt, tuy nhiên vẫn cần phân biệt giữa hai kiểu chuỗi này vì có những thuật toán khác nhau thích hợp cho từng kiểu chuỗi và cũng do các chuỗi nhị phân hay nảy sinh một cách tự nhiên trong nhiều ứng dụng. Ví dụ, một vài hệ đồ họa máy tính biểu diễn ảnh như các chuỗi nhị phân (Quyển sách này được in trên một hệ như vậy : trang hiện tại đã được biểu diễn như một chuỗi nhị phân gồm hàng triệu bits).

Theo một nghĩa nào đó, các chuỗi văn bản là các đối tượng hoàn toàn khác biệt với các chuỗi nhị phân, vì chúng được tạo ra bởi các ký tự từ một bảng chữ cái lớn. Theo một nghĩa khác, thì hai kiểu chuỗi là tương đương, vì mỗi ký tự văn bản có thể được biểu diễn bởi tám bits nhị phân, còn một chuỗi nhị phân có thể được xem như một chuỗi văn bản bằng cách xem các nhóm tám bits nhị phân như là các ký tự. Chúng ta sẽ thấy rằng kích thước của bảng chữ cái từ đó các ký tự được lấy ra để tạo nên một chuỗi

sẽ là một thành tố quan trọng trong việc thiết kế các thuật toán xử lý chuỗi.

Một phép toán cơ bản trên chuỗi là đối sánh mẫu (Pattern Matching): Cho trước một chuỗi văn bản có độ dài N và một mẫu có độ dài M, hãy tìm sự xuất hiện của mẫu trong văn bản (Ta sử dụng thuật ngữ "văn bản" ngay cả khi nói tới một dãy các giá trị 0-1 hay một kiểu chuỗi đặc biệt khác). Hầu hết các thuật toán cho bài toán này có thể dễ dàng mở rộng để tìm tất cả các xuất hiện của mẫu trong văn bản, vì chúng sẽ quét qua toàn bộ văn bản một cách tuần tự và có thể được bắt đầu trở lại ở điểm ngay sau điểm bắt đầu của một lần xuất hiện để tìm tiếp lần xuất hiện kế của mẫu.

Bài toán đối sánh mẫu có thể được đặc trưng như một bài toán tìm kiếm, trong đó mẫu được xem như khoá, tuy nhiên các thuật toán tìm kiếm mà ta đã nghiên cứu không áp dụng được một cách trực tiếp vì mẫu có thể dài và do nó trải (lines up) trên văn bản theo một cách không biết trước được. Đây là một bài toán thú vị : nhiều thuật toán rất khác nhau (và cũng rất bất ngờ) chỉ mới được phát hiện gần đây không những đã cung cấp một loạt các phương pháp thực tế hữu ích mà còn minh họa cho một vài kỹ thuật thiết kế thuật toán cơ sở.

MỘT ÍT VỀ LỊCH SỬ

Các thuật toán ta đã xem xét có một lịch sử thú vị: chúng ta sẽ tóm tắt nó ở đây để từ đó giúp ta có thể đặt các phương pháp khác nhau này vào đúng bối cảnh lịch sử của chúng.

Có một thuật toán thô thiển ("brute-force") hiển nhiên để xử lý chuỗi đang được sử dụng rộng rãi. Mặc dù thời gian thực hiện tồi nhất của nó tỉ lệ với tích MN, nhưng trong nhiều ứng dụng thực tế các chuỗi phát sinh ra thường có thời gian xử lý thực sự luôn tỉ lệ với tổng M+N.

Hơn thế nữa, nó thích hợp với cấu trúc của hầu hết các hệ máy tính, vì thế một bản chương trình tối ưu sẽ cung cấp một "chuẩn mực" khó bị đánh bại bởi một thuật toán thông minh hơn.

Trong năm 1970, S.A. Cook đã chứng minh một kết quả lý thuyết về một loại máy trừu tượng mà nó suy ra sự tồn tại của một thuật toán để giải bài toán đối sánh mẫu, có thời gian tỉ lệ với M+N trong trường hợp xấu nhất. D.E. Knuth & V.R. Pratt đã theo đuổi một cách kiên trì kiến trúc mà Cook đã dùng để chứng minh cho định lý của ông ấy (định lý đó không có dự định biến thành thực tế) và đã nhận được một thuật toán mà nó có thể được tinh chế thành một thuật toán thực tiễn tương đối đơn giản. Có vẻ như đây là một trường hợp hiếm hoi khi mà một kết quả lý thuyết lại có khả năng áp dụng được ngay (và không dự kiến trước). Nhưng hoá ra là J.H. Morris đã thực sự khám phá ra cùng một thuật toán, được xem như là lời giải cho một bài toán thực tế rắc rối mà ông gặp phải khi cài đặt một trình soạn thảo văn bản. Tuy nhiên, sự kiện phát sinh cùng một thuật toán từ hai cách tiếp cận khác nhau như vậy đã làm tăng thêm sự tin tưởng vào thuật toán, với tư cách là một lời giải cơ sở cho bài toán.

Knuth, Morris và Pratt đã không giới thiệu thuật toán của họ mãi cho đến năm 1976, và trong thời gian đó R.S.Boyer và J.S.Moore (và độc lập một mình là W.Gosper) đã khám phá ra một thuật toán nhanh hơn nhiều trong nhiều ứng dụng, vì nó thường chỉ kiểm tra một phần các ký tự trong chuỗi văn bản. Nhiều trình soạn thảo văn bản sử dụng thuật toán này để đạt được sự rút ngắn đáng kể về thời gian trả lời trong khi tìm kiếm chuỗi.

Cả thuật toán của Knuth-Morris-Pratt lẫn của Boyer-Moore đều cần đến một chút xử lý phức tạp trên mẫu, khiến cho thuật toán trở nên khó hiểu và do đó đã hạn chế phạm vi sử dụng của chúng (Thật vậy, chuyện là có một lập trình viên hệ thống nào đó đã thấy rằng thuật toán của Morris là quá khó hiểu và đã thay nó bằng cài đặt của thuật toán brute-force).

Trong năm 1980, R.M.Karp và M.O.Rabin đã quan sát thấy rằng bài toán này không khác lắm so với bài toán tìm kiếm chuẩn như người ta đã tưởng, và đã đi đến một thuật toán đơn giản gần như thuật toán brute-force, có thời gian thi hành luôn tỉ lệ với M+N. Hơn thế nữa, thuật toán của họ mở rộng dễ dàng cho các mẫu và văn bản 2 chiều, do đó khiến cho nó trở nên hữu ích hơn so với các thuật toán còn lại trong việc xử lý ảnh.

THUẬT TOÁN BRUTE-FORCE

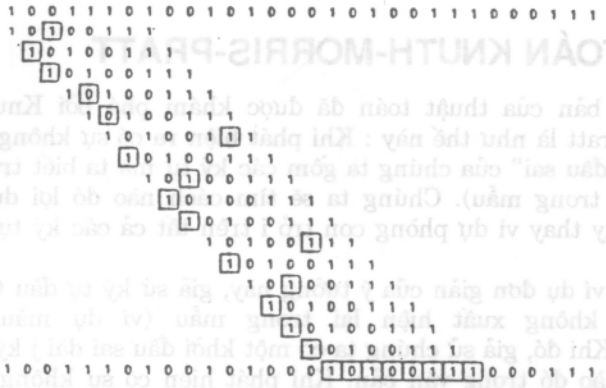
Phương pháp đối sánh mẫu hiển nhiên nảy sinh ngay trong đầu là chỉ việc kiểm tra từng vị trí trong văn bản ở đó mẫu có thể khớp được, cho đến khi nào chúng khớp nhau thực sự. Chương trình sau đây sẽ đi tìm sự xuất hiện đầu tiên của mẫu p[1..M] trong một chuỗi văn bản a[1..N].

```
function brutesearch : integer;
   var i,j:integer;
   begin
   i:=1; j:=1;
   repeat
      if a[i] = p[j]
      then begin i:=i+1; j:=j+1 end
      else begin i:=i-j+2; j:=1 end;
   until (j>M) or (i>N);
   if j>M
      then brutesearch := i-M
      else brutesearch := i
   end;
```

Chương trình lưu giữ một con trỏ (i) trong văn bản và một con trỏ khác (j) trong mẫu. Nếu chúng trỏ tới những ký tự khớp nhau thì cả hai con trỏ đều được tăng lên. Nếu gặp cuối mẫu (j>M) thì có nghĩa là đã tìm thấy một sự trùng khớp. Nếu i và j trỏ tới những ký tự không khớp nhau, thì j được đặt lại để trỏ tới nơi bắt đầu của mẫu và i được đặt lại tương ứng với việc di chuyển mẫu sang phải một vị trí để thực hiện việc so khớp lại. Khi đã tới cuối văn bản (i>N) thì có nghĩa là không còn có sự trùng khớp nào nữa. Nếu mẫu không xuất hiện trong văn bản, thì giá trị N+1 được trả về.

Trong một áp dụng soạn thảo văn bản, vòng lặp trong của chương trình này hiếm khi được lặp lại và thời gian chạy thì gần như tỉ lệ với số ký tự văn bản đã được kiểm tra. Ví dụ, giả sử ta đang tìm mẫu "STING" trong chuỗi văn bản "A STRING SEARCHING EXAMPLE CONSISTING OF ..."

Khi đó lệnh j:=j+1 chỉ được thực hiện 4 lần (chỉ một lần cho mỗi S, 2 lần cho ST đầu tiên) trước khi gặp một sự trùng khớp thực sự.



Hình 19.1 *Phép tìm chuỗi Brute-force trong văn bản nhị phân*

Mặt khác, việc tìm kiếm brute-force có thể là rất chậm đối với một số mẫu nào đó, ví dụ nếu văn bản là nhị phân (2 ký tự), mà nó có thể xuất hiện trong các ứng dụng xử lý ảnh và lập trình hệ thống. Hình 19.1 cho thấy những gì xảy ra khi thuật toán được dùng để tìm mẫu 10100111 trong một chuỗi văn bản nhị phân dài. Mỗi dòng (trừ dòng cuối là dòng trùng khớp) không có hay có nhiều ký tự khớp với mẫu, nhưng lại theo sau bởi một ký tự không khớp. Những dòng này gọi là "các khởi đầu sai", xảy ra khi thử tìm mẫu; một mục tiêu hiển nhiên trong việc thiết kế thuật toán là thử hạn chế số lần và chiều dài của các "khởi đầu sai" này.

Tính chất 19.1 *Phép tìm chuỗi brute-force có thể cần khoảng NM phép so sánh ký tự.*

Trường hợp xấu nhất là khi cả mẫu lẫn văn bản, tất đều là các số 0, được theo sau bởi một số 1. Khi đó với mỗi vị trí trong N-M+1 vị trí có thể khớp, thì tất cả các ký tự trong mẫu đều phải được kiểm tra trên văn bản, với một chi phí tổng cộng là M(N-M+1). Thường thì M rất nhỏ so với N, như thế tổng số phép so sánh là khoảng NM.

Các chuỗi thoái hoá như vậy không thích hợp trong văn bản tiếng Anh (hay Pascal), nhưng ta có thể bắt gặp chúng khi xử lý các văn bản nhị phân, vì vậy chúng ta sẽ đi tìm các thuật toán tốt hơn.

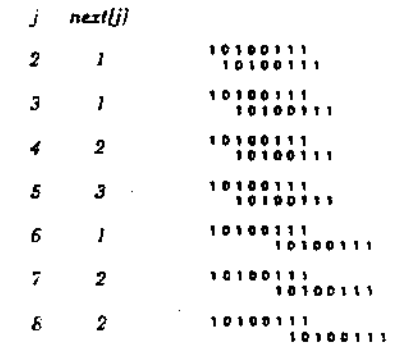
THUẬT TOÁN KNUTH-MORRIS-PRATT

Ý tưởng cơ bản của thuật toán đã được khám phá bởi Knuth, Morris và Pratt là như thế này : Khi phát hiện ra có sự không ăn khớp, "khởi đầu sai" của chúng ta gồm các ký tự mà ta biết trước (vì chúng ở trong mẫu). Chúng ta sẽ tìm cách nào đó lợi dụng thông tin này thay vì dự phòng con trỏ i trên tất cả các ký tự đã biết đó.

Với một ví dụ đơn giản của ý tưởng này, giả sử ký tự đầu tiên trong mẫu không xuất hiện lại trong mẫu (ví dụ mẫu là 10000000). Khi đó, giả sử chúng ta có một khởi đầu sai dài j ký tự ở một chỗ nào đó trong văn bản. Khi phát hiện có sự không ăn khớp, ta biết một sự kiện là j ký tự đã khớp nhau, mà ta không cần "dự phòng" con trỏ văn bản i, vì không có ký tự nào trong số j-1 ký tự trong văn bản có thể trùng với ký tự đầu tiên trong mẫu. Sự sửa đổi này có thể được cài đặt bằng cách thay thế i:=i-j+2 trong chương trình ở trên bởi i:=i+1. Tác động thực tế của sự thay đổi này bị hạn chế do một mẫu đặc biệt như thế thường không hay xảy ra, nhưng ý tưởng là sự suy nghĩ đáng giá về nó và thuật toán Knuth-Morris-Pratt là một sự tổng quát hoá của suy nghĩ đó. Thật đáng ngạc nhiên, là luôn luôn có thể sắp xếp sao cho con trỏ i không bao giờ bị giảm đi.

Việc nhảy ngược lại đầu mẫu khi phát hiện ra một sự không trùng khớp đã mô tả trong đoạn trước sẽ không chạy được khi chính bản thân mẫu có thể khớp nhau ở điểm không ăn khớp. Ví dụ, khi tìm 10100111 trong dãy 1010100111 đầu tiên ta phát hiện ra có sự không ăn khớp ở ký tự thứ năm, nhưng tốt hơn là ta nên dự phòng tới ký tự thứ ba để tiếp tục việc tìm kiếm, vì nếu không thì ta có thể bỏ sót một dãy khớp nhau. Nhưng ta có thể giải quyết trước thời gian đúng những gì phải làm, do nó chỉ phụ thuộc vào mẫu, như minh họa trong hình 19.2.

Mảng next[1..M] sẽ được dùng để xác định phải dự phòng một khoảng bao nhiêu khi phát hiện sự không ăn khớp. Tưởng tượng là ta để trượt một bản sao của j-1 ký tự đầu tiên của mẫu trên chính nó, từ trái sang phải, bắt đầu từ ký tự đầu tiên của bản sao trên ký tự thứ hai của mẫu và ngừng khi tất cả các ký tự gối khớp nhau (hay không có ký tự nào khớp nhau cả). Những ký tự gối nhau này xác định vị trí kế tiếp có thể có sự trùng khớp, nếu phát hiện ra có



Hình 19.2 *Các vị trí bắt đầu lại đối với cách tìm Knuth-Morris-Pratt*

sự không khớp ở p[j]. Khoảng cách để dự phòng trong mẫu (next[j]) chính xác là 1 cộng với số ký tự gối nhau. Đặc biệt, với j>1, giá trị của next[j] là số k lớn nhất < j mà k-1 ký tự đầu tiên của mẫu khớp với k-1 ký tự cuối cùng của j-1 ký tự đầu tiên của mẫu. Như chúng ta sẽ thấy, tiện hơn là định nghĩa next[1] là 0.

Mảng next này cho ta ngay một cách để giới hạn (thực ra, như chúng ta sẽ thấy, là bỏ đi) việc "dự phòng" của con trỏ văn bản i, như đã bàn ở trên. Khi i và j trỏ tới những ký tự không khớp nhau (kiểm tra xem có sự trùng khớp mẫu bắt đầu ở vị trí i-j+1 trong chuỗi văn bản hay không), sau đó vị trí kế tiếp có khả năng trùng khớp mẫu sẽ bắt đầu ở vị trí i-next[j]+1. Nhưng do định nghĩa của bảng next, next[j]-1 ký tự đầu tiên ở vị trí đó trùng với next[j]-1 ký

```
function kmpsearch : integer;
  var i,j:integer;
  begin i:=1; j:=1; initnext;
  repeat
    if (j=0) or (a[i]=p[j])
    then begin i:=i+1; j:=j+1 end
    else begin      j:=next[j] end;
  until (j>M) or (i>N);
  if j>M
  then kmpsearch := i-M
  else kmpsearch := i;
  end;
```

tự đầu của mẫu, vì vậy không cần dự phòng con trỏ i: có thể để cho con trỏ i không thay đổi và đặt con trỏ j tới next[j], như trong chương trình trên.

Khi j=1 và a[i] không khớp với mẫu, không có sự gối nhau, như thế ta muốn tăng i và đặt j cho trỏ tới nơi bắt đầu của mẫu. Điều này đạt được bằng cách định nghĩa next[1] là 0, để cho j được đặt về 0; sau đó i được tăng lên và j được đặt về 1 ở lần lặp kế (để thực hiện được mẹo này, mảng chứa mẫu phải được khai báo bắt đầu từ chỉ số 0, vì nếu không, Pascal chuẩn sẽ phàn nàn về lỗi "Chỉ số mảng nằm ngoài phạm vi" ("Subscript out of range") khi j=0 mặc dù nó không thực sự phải truy xuất p[0] để xác định chân trị của or). Về chức năng, chương trình này giống như chương trình brutesearch, nhưng nó có khả năng chạy nhanh hơn đối với các mẫu có tính tự lặp lại cao.

Ta chỉ còn phải tính bảng next. Chương trình của phần này ngắn nhưng rắc rối: về cơ bản, nó giống với chương trình ở trên, ngoại trừ là nó được dùng để so khớp mẫu với chính nó :

```
procedure initnext;
  var i,j:integer;
  begin
  i:=1; j:=0; next[1]:=0;
  repeat
    if (j=0) or (p[i]=p[j])
    then begin i:=i+1; j:=j+1; next[i]:=j end
    else begin j:=next[j] end;
  until i>=M;
  end;
```

Chỉ sau khi i và j được tăng lên, j-1 ký tự đầu của mẫu đã được xác định là khớp với các ký tự nằm trong phạm vi p[i-j+1..i-1], j-1 ký tự cuối trong i-1 ký tự đầu của mẫu. Và đây là j lớn nhất có tính chất này, vì nếu không thì ta có thể bỏ sót mất một "sự trùng khớp có thể" của mẫu với chính nó. Vì vậy j đúng là giá trị sẽ được gán vào next[i].

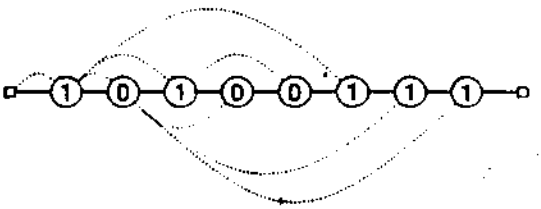
Một cách thú vị để xem thuật toán này là coi như mẫu không đổi, sao cho bảng next có thể góp phần mình vào chương trình. Ví dụ, chương trình sau hoàn toàn tương đương với chương trình trên, cho mẫu mà ta đang xét, nhưng hiệu quả hơn nhiều :

```
    i:=0;
0: i:=i+1;
1: if a[i] <> '1' then goto 0; i:=i+1;
2: if a[i] <> '0' then goto 1; i:=i+1;
3: if a[i] <> '1' then goto 1; i:=i+1;
4: if a[i] <> '0' then goto 2; i:=i+1;
5: if a[i] <> '0' then goto 3; i:=i+1;
6: if a[i] <> '1' then goto 1; i:=i+1;
7: if a[i] <> '1' then goto 2; i:=i+1;
8: if a[i] <> '1' then goto 2; i:=i+1;
    search := i-8;
```

Các nhãn goto trong chương trình này tương ứng một cách chính xác với bảng next. Thực vậy, chương trình initnext để tính bảng next ở trên có thể dễ dàng sửa lại để đưa ra thành chương trình này ! Để tránh phải kiểm tra i>N mỗi lần i tăng lên, ta giả sử chính mẫu được chứa ở cuối của văn bản, như một người lính canh, trong mảng a[N+1..N+M]. (Việc tối ưu hoá này cũng có thể được áp dụng cho bản cài đặt chuẩn). Đây là một ví dụ đơn giản của một "trình biên dịch tìm-chuỗi" : cho trước một mẫu, ta có thể sinh ra một chương trình rất hiệu quả để dò mẫu đó trong một chuỗi văn bản dài tùy ý. Chúng ta sẽ thấy tính khái quát hoá của khái niệm này trong hai chương kế.

Chương trình ở trên chỉ dùng một vài phép toán rất cơ bản để giải quyết bài toán tìm kiếm chuỗi. Điều này có nghĩa là chương trình có thể dễ dàng được mô tả theo các thuật ngữ của một mô hình máy rất đơn giản được gọi là một máy trạng-thái-hữu-hạn (finite-state machine). Hình 19.3 minh hoạ máy trạng thái hữu hạn cho chương trình ở trên.

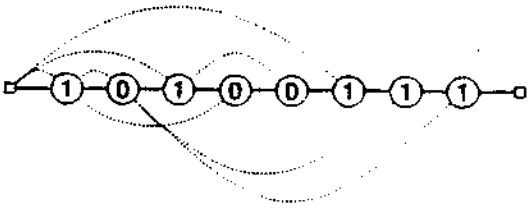
Máy gồm các trạng thái (được chỉ ra bởi các số khoanh tròn) và



Hình 19.3 *Máy trạng thái hữu hạn cho thuật toán Knuth-Morris-Pratt*

các chuyển tiếp (được chỉ ra bởi các đường). Mỗi trạng thái có hai chuyển tiếp rời khỏi nó : một chuyển tiếp khớp (đường tô đặc, đi sang phải) và một chuyển tiếp không khớp (đường nét rời, đi sang trái). Các trạng thái là nơi máy thực thi lệnh; các chuyển tiếp là các lệnh goto. Khi trạng thái được gán nhãn "x", máy chỉ có thể thực hiện một lệnh, "nếu ký tự hiện hành là x thì quét qua nó và lấy chuyển tiếp khớp, nếu không thì lấy chuyển tiếp không khớp". "Quét qua" một ký tự nghĩa là lấy ký tự kế trong chuỗi làm "ký tự hiện hành"; máy quét qua các ký tự khi nó khớp với chúng. Có hai ngoại lệ đối với quy luật này : trạng thái đầu luôn lấy một chuyển tiếp khớp và quét tới ký tự kế (chủ yếu điều này ứng với việc dò ra lần xuất hiện đầu tiên của ký tự đầu tiên trong mẫu) và trạng thái cuối cùng là một trạng thái "dừng" chỉ ra rằng đã tìm thấy có sự trùng khớp. Trong chương kế ta sẽ thấy làm thế nào dùng một máy tương tự như vậy (nhưng mạnh hơn) để giúp phát triển một thuật toán đối sánh mẫu mạnh hơn nhiều.

Các độc giả nhanh trí có thể nhận thấy là vẫn còn có chỗ để nâng cấp thuật toán này, do thuật toán không chú ý vào ký tự đã gây nên sự không ăn khớp. Ví dụ, giả sử văn bản của chúng ta bắt đầu bằng 1011 và ta đang đi tìm mẫu là 10100111. Sau khi khớp 101, ta thấy có sự không khớp ở ký tự thứ tư; ở điểm này bảng next bảo đi kiểm tra ký tự thứ hai của mẫu dựa trên ký tự thứ tư của văn bản, vì trên cơ sở khớp 101, ký tự đầu của mẫu có thể ứng (line up) với ký tự thứ ba của văn bản (nhưng ta không phải so sánh những ký tự này vì ta đã biết cả hai đều là số 1). Tuy nhiên, không thể có một sự ăn khớp ở đây: từ sự không trùng khớp, ta biết rằng ký tự kế tiếp trong văn bản là khác 0, như được yêu cầu bởi mẫu. Có một cách khác để thấy điều này là nhìn vào bản chương trình với bảng next "được tham dự" (wired in) : ở nhãn số 4, ta nhảy tới 2 nếu a[i] khác 0, nhưng ở nhãn 2 ta nhảy tới 1 nếu



Hình 19.4

Máy trạng thái hữu hạn Knuth-Morris-Pratt (đã được nâng cấp)

a[i] khác 0. Tại sao không nhảy trực tiếp tới 1 ? Hình 19.4 cho thấy bản nâng cấp của máy trạng thái hữu hạn đối với ví dụ của chúng ta.

May mắn là ta dễ dàng đưa sự thay đổi này vào trong thuật toán. Ta chỉ cần thay lệnh next[i]:=j trong chương trình initnext bởi

---

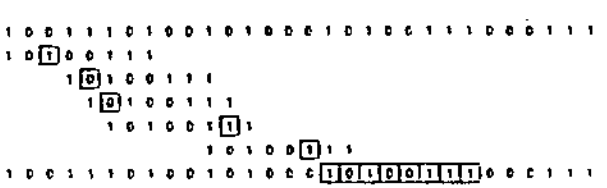
if *p[j] <> p[i]* **then** *next[i] := j* **else** *next[i] := next[j];*

Vì ta đang tiến hành từ trái sang phải, gía trị cần thiết của next đã được tính trước rồi, vì vậy ta chỉ việc dùng nó mà thôi.

Tính chất 19.2 *Phép tìm kiếm chuỗi Knuth-Morris-Pratt không bao giờ dùng nhiều hơn M+N phép so sánh ký tự.*

Tính chất này được minh họa trong hình 19.5, và từ chương trình ta thấy rõ một điều là : hoặc là ta tăng j hoặc là ta đặt lại nó từ bảng next hầu như chỉ một lần cho mỗi i.

Hình 19.5 cho thấy là phương pháp này đặc biệt dùng ít phép so sánh hơn nhiều so với phương pháp brute-force cho ví dụ nhị phân của chúng ta. Tuy nhiên thuật toán Knuth-Morris-Pratt không nhanh hơn đáng kể so với phương pháp brute-force trong nhiều ứng dụng thực tế, do ít có ứng dụng nào hàm chứa việc tìm kiếm các mẫu có tính tự lặp lại cao trong một văn bản cũng có tính tự lặp lại cao. Tuy nhiên phương pháp này có một thuận lợi thực tế quan trọng: nó tiến hành tuần tự qua dãy nhập và không bao giờ dự phòng trong dãy nhập. Điều này khiến cho nó trở nên thuận tiện khi dùng trên một tập tin lớn được đọc từ một thiết bị ngoại vi nào đó (các thuật toán yêu cầu dự phòng sẽ phải cần một chút phép trữ đệm (buffering) phức tạp trong trường hợp này).

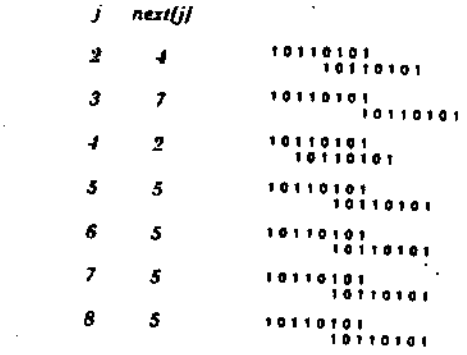


Hình 19.5 *Phép tìm chuỗi Knuth-Morris-Pratt trong văn bản nhị phân*

THUẬT TOÁN BOYER-MOORE

Nếu việc "dự phòng" không khó khăn, thì ta có thể phát triển một phương pháp tìm chuỗi nhanh hơn đáng kể bằng cách quét mẫu từ phải sang trái khi thử so khớp nó với văn bản. Khi tìm mẫu 10100111, nếu ta thấy có những sự trùng khớp trên các ký tự thứ 8, thứ 7 và thứ 6 nhưng không khớp ở ký tự thứ 5, thì ta có thể cho trượt ngay mẫu sang phải 7 vị trí, và kiểm ký tự thứ 5 kế tiếp, vì việc so khớp từng phần đã tìm ra được 111, mà nó có thể xuất hiện ở đâu đó trong mẫu. Dĩ nhiên, mẫu ở cuối thường xuất hiện ở đâu đó trong mẫu, như thế ta cần có một bảng next như trên.

Một bảng next cho trường hợp đi từ phải sang trái đối với mẫu 10110101 được minh họa trong hình 19.6 : trong trường hợp này next[j] là số vị trí ký tự mà ta có thể dịch mẫu sang phải, nếu cho trước là có một sự không trùng khớp trong lần quét từ phải-sang-trái đã xảy ra trên ký tự thứ j, tính từ bên phải, ở trong mẫu. Điều này được thấy như lúc trước, bằng cách cho trượt một bản sao của mẫu trên j-1 ký tự cuối của chính nó từ trái sang phải, bắt đầu từ ký tự kế chót của bản sao ứng với ký tự chót của mẫu và ngừng khi tất cả các ký tự gối nhau trùng khớp lên nhau (cũng đưa vào ký tự đã gây ra sự không ăn khớp). Ví dụ, next[3]=7 vì, nếu có một sự trùng khớp của hai ký tự cuối và sau đó là không khớp trong một lần quét từ phải sang trái, thì 001 phải bị bắt gặp



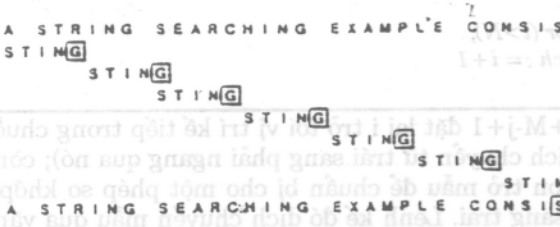
Hình 19.6 *Các vị trí bắt đầu lại cho phép tìm Boyer-Moore*

trong văn bản; dãy này không xuất hiện trong mẫu, ngoại trừ có khả năng số 1 ứng với ký tự đầu tiên trong mẫu, như thế ta có thể trượt đi bảy vị trí sang bên phải.

Từ đó dẫn đến một chương trình hoàn toàn giống với cài đặt của phương pháp Knuth-Morris-Pratt ở trên. Ta sẽ không khai thác chương trình này chi tiết hơn nữa vì có một cách hoàn toàn khác để nhảy qua (skip) các ký tự với việc quét mẫu từ phải sang trái tốt hơn hẳn trong nhiều trường hợp.

Ý tưởng là quyết định xem sẽ làm gì kế tiếp trên cơ sở của ký tự đã gây ra sự không trùng khớp trong văn bản cũng như trong mẫu. Bước tiền xử lý là quyết định xem với mỗi ký tự có thể xuất hiện trong văn bản, chúng ta sẽ làm gì nếu ký tự đó là ký tự gây ra sự không ăn khớp. Nhận thức đơn giản nhất của điều này dẫn ngay tới một chương trình thật hữu ích.

Hình 19.7 minh họa phương pháp này trên văn bản mẫu đầu tiên của chúng ta. Tiến hành từ phải sang trái để so khớp mẫu, đầu tiên ta kiểm tra G trong mẫu dựa trên R (ký tự thứ 5) trong văn bản. Không những các ký tự này không khớp nhau, mà ta còn có thể chú ý là R không xuất hiện ở một chỗ nào trong mẫu hết, như thế ta có thể trượt nó qua R. Phép so sánh kế tiếp là cho ký tự G trong mẫu dựa trên ký tự thứ năm theo sau R (chữ S trong SEARCHING). Lúc này ta có thể trượt mẫu sang phải cho đến khi S của nó khớp với S của mẫu trong văn bản. Rồi đến G trong mẫu được so với C trong SEARCHING, mà nó không xuất hiện trong mẫu, như thế mẫu có thể được đẩy trượt đi thêm 5 vị trí nữa sang phải. Sau 3 lần nhảy 5-ký-tự nữa thì ta gặp T trong CONSISTING, ở vị trí đó ta sắp mẫu sao cho T của nó khớp với T trong văn bản



Hình 19.7 *Phép tìm chuỗi Boyer-Moore sử dụng heuristic ký tự bất sánh*

và tìm được sự trùng khớp hoàn toàn. Phương pháp này đưa chúng ta tới đúng vị trí khớp nhau với cái giá là chỉ cần kiểm tra qua 7 ký tự trong văn bản (và thêm 5 ký tự nữa để kiểm tra sự trùng khớp).

Thuật toán "ký-tự-không-khớp" (mismatched-character) này thật dễ cài đặt. Đơn giản là nâng cấp phép quét mẫu brute-force từ phải-sang-trái để khởi động một mảng skip mà nó cho biết là ứng với mỗi ký tự trong bảng chữ cái, ta phải nhảy qua một khoảng bao nhiêu nếu ký tự đó xuất hiện trong văn bản và gây ra sự không ăn khớp trong khi tìm chuỗi. Phải có một điểm vào (entry) trong mảng skip cho mỗi ký tự có thể xảy ra trong văn bản : để đơn giản, giả sử rằng ta đã có một hàm function index(c:char):integer; mà nó trả về 0 đối với các khoảng trắng và i đối với ký tự thứ i của bảng chữ cái; ta cũng giả sử rằng có một thủ tục procedure initskip mà nó khởi đông giá trị mảng skip là M đối với các ký tự không có trong mẫu và sau đó, với j chạy từ 1 tới M, đặt skip[index(p[j])] là M-j. Cài đặt thì đơn giản :

```
function mischarsearch : integer;
   var i,j:integer;
   begin
   i:=M; j:=M; initskip;
   repeat
      if a[i]=p[j]
      then begin i:=i-1; j:=j-1 end
      else
         begin
               if M-j+1  skip[index(a[i])]
               then i:=i+M-j+1
               else i:=i+skip[index(a[i])];
               j:=M;
         end;
   until (j<1) or (i>N);
   mischarsearch := i+1
   end;
```

Lệnh i:=i+M-j+1 đặt lại i trỏ tới vị trí kế tiếp trong chuỗi văn bản (vì mẫu dịch chuyển từ trái sang phải ngang qua nó); còn lệnh j:=M đặt lại con trỏ mẫu để chuẩn bị cho một phép so khớp từng ký tự từ phải sang trái. Lệnh kế đó dịch chuyển mẫu qua văn bản, nếu được trao điều khiển. Đối với mẫu STING, điểm vào trong

mảng skip cho G sẽ là 0, điểm vào cho N sẽ là 1, điểm vào cho I sẽ là 2, điểm vào cho T sẽ là 3, điểm vào cho S sẽ là 4, và các điểm vào cho tất cả các ký tự khác sẽ là 5. Vì vậy, ví dụ khi bắt gặp một ký tự S trong một lần tìm từ phải sang trái, con trỏ i sẽ được tăng thêm 4 sao cho cuối của mẫu được sắp thành hàng bốn vị trí về bên phải của S (và kết quả là S trong mẫu ứng với S trong văn bản). Nếu có nhiều hơn một ký tự S trong mẫu, ta mong muốn dùng được ký tự S nằm tận cùng bên phải đối với cách tính này: vì vậy mảng skip được tạo ra bằng cách quét từ trái sang phải.

Boyer và Moore đã đề nghị kết hợp cả hai phương pháp để quét mẫu từ phải sang trái mà ta đã tóm tắt, rồi chọn ra skip nào lớn hơn trong hai skip được yêu cầu.

TÍNH CHẤT 19.3 *Phép tìm chuỗi Boyer-Moore không bao giờ dùng nhiều hơn M+N phép so sánh ký tự, và dùng khoảng N/M bước nếu bảng chữ cái không nhỏ và mẫu không dài.*

Thuật toán là tuyến tính trong trường hợp xấu nhất theo cùng cách thức như phương pháp Knuth-Morris-Pratt (cài đặt đã được cho ở trên mà nó chỉ thực hiện một trong hai tính chất heuristic Boyer-Moore, là không tuyến tính). Kết quả N/M của "trường hợp trung bình" có thể được chứng minh đối với các mô hình chuỗi ngẫu nhiên khác nhau, nhưng các mô hình này có vẻ không thực tế, do đó ta sẽ bỏ qua các chi tiết. Trong nhiều trường hợp thực tế, đúng ra là tất cả các ký tự của bảng chữ cái không xuất hiện ở đâu cả trong mẫu, ngoại trừ một số ít là có trong mẫu, do đó mỗi phép so sánh sẽ dẫn tới việc nhảy qua M ký tự, và điều này dẫn đến kết quả đã được phát biểu.

Thuật toán "ký tự không khớp" rõ ràng sẽ không giúp ích gì nhiều cho các chuỗi nhị phân, vì chỉ có hai khả năng để cho các ký tự trở thành không khớp (và cả hai khả năng này đều có thể nằm trong mẫu). Tuy nhiên các bit có thể được nhóm lại với nhau để tạo nên các "ký tự" mà nó có thể được dùng y như trên. Nếu ta lấy b bits ở một thời điểm, thì ta cần một bảng skip với 2^b đầu vào. Giá trị của b sẽ được chọn đủ nhỏ để cho bảng này không quá lớn nhưng đủ lớn để cho hầu hết các đoạn b-bit của văn bản không nằm trong mẫu. Đặc biệt, có M-b+1 đoạn b-bit khác nhau trong

mẫu (một đoạn bắt đầu ở mỗi vị trí bit từ 1 đến M-b+1), như thế ta muốn M-b+1 sẽ nhỏ hơn đáng kể so với 2^b. Ví dụ, nếu ta lấy b là khoảng lg(4M), thì bảng skip sẽ có hơn 3/4 được đổ đầy với M đầu vào. b cũng phải nhỏ hơn M/2, vì nếu không ta có thể bỏ sót toàn bộ mẫu nếu nó bị tách ra giữa hai đoạn văn bản b-bit.

THUẬT TOÁN RABIN-KARP

Một cách tiếp cận brute-force cho việc tìm kiếm chuỗi mà ta đã không xem xét ở trên là khai thác một vùng nhớ lớn bằng cách xem mỗi đoạn M-ký tự có thể có của văn bản như là một khoá (key) trong một bảng băm chuẩn. Nhưng không cần thiết phải giữ một bảng băm tổng thể, vì bài toán được cài đặt sao cho chỉ một khoá là đang được tìm kiếm; việc mà ta cần làm là đi tính hàm băm cho M ký tự từ văn bản vì nó chỉ đơn giản là kiểm tra xem chúng có bằng với mẫu hay không. Rabin và Karp đã tìm ra một phương pháp dễ dàng để giải quyết trở ngại này đối với hàm băm ta đã dùng trong chương 16: h(k) = k mod q, ở đây q (kích thước bảng) là một số nguyên tố lớn. Trong trường hợp này, không có gì được chứa trong bảng băm, vì vậy q có thể được cho giá trị rất lớn.

Phương pháp này dựa trên việc tính hàm băm cho vị trí i trong văn bản, cho trước giá trị tại vị trí i-1 của nó, và suy ra hoàn toàn trực tiếp từ công thức toán học. Giả sử rằng ta dịch M ký tự thành số bằng cách nén chúng lại với nhau trong một từ (word) của máy, mà ta xem nó như một số nguyên. Điều này ứng với việc ghi các ký tự như các con số trong một hệ thống cơ số d, ở đây d là số ký tự có thể có. Vì vậy số ứng với a[i..i+M-1] là

$$x = a[i]d^{M-1} + a[i+1]d^{M-2} + \ldots + a[i+M-1]$$

và có thể giả sử rằng ta biết giá trị của h(x) = x mod q. Nhưng dịch (shift) một vị trí sang phải trong văn bản tương ứng với việc thay x bởi

$$(x - a[i]d^{M-1})d + a[i+M]$$

Một tính chất cơ bản của phép toán mod là ta có thể thực hiện nó bất kỳ lúc nào trong các phép toán này và vẫn nhận được cùng câu trả lời. Cách khác, nếu ta lấy phần dư khi chia cho q sau mỗi

một phép toán số học (để giữ cho các số mà ta đang gặp là nhỏ), thì ta sẽ nhận được cùng câu trả lời như thể ta đã thực hiện tất cả các phép toán số học, sau đó lấy phần dư khi chia cho q.

Điều này dẫn tới một thuật toán đối sánh mẫu rất đơn giản được cài đặt dưới đây, Chương trình giả định dùng cùng hàm index như trên, nhưng dùng d=32 để cho hiệu quả (các phép nhân có thể được cài đặt như các phép dịch bit)

```
function rksearch : integer;
   const q=33554393; d=32;
   var h1,h2,dM,i : integer;
   begin
   dM:=1;
   for i:=1 to M-1 do dM:=(d*dM) mod q;
   h1:=0;
   for i:=1 to M do h1:=(h1*d+index(p[i])) mod q;
   h2:=0;
   for i:=1 to M do h2:=(h2*d+index(a[i])) mod q;
   i:=1;
   while (h1<>h2) and (i<=N-M ) do
      begin
      h2:=(h2+d*q-index(a[i])*dM) mod q;
      h2:=(h2*d+index(a[i+M])) mod q;
      i:=i+1;
      end;
   rksearch:=i;
   end;
```

Đầu tiên chương trình tính giá trị h1 cho mẫu, sau đó tới giá trị h2 cho M ký tự đầu tiên của văn bản (nó cũng tính giá trị của d^{M-1} mod q trong biến dM). Sau đó nó tiến hành công việc qua chuỗi văn bản, dùng đến kỹ thuật ở trên để tính hàm băm cho M ký tự, bắt đầu ở vị trí i đối với mỗi i và so sánh từng giá trị băm mới với h1. Số nguyên tố q được chọn càng lớn càng tốt, nhưng đủ nhỏ sao cho (d+1)*q không gây ra tràn (overflow) : điều này cần ít phép toán mod hơn nếu ta dùng số nguyên tố lớn nhất biểu diễn được (một giá trị d*q phụ trợ được cộng thêm vào trong khi tính h2 để bảo đảm rằng mọi đại lượng vẫn còn là dương để cho phép toán mod có thể thực hiện được).

TÍNH CHẤT 19.4 *Phép đối sánh mẫu Rabin-Karp gần như là tuyến tính.*

Thuật toán này hiển nhiên thực hiện theo thời gian tỉ lệ với M+N, nhưng chú ý là nó chỉ thực sự đi tìm một vị trí trong văn bản có cùng giá trị băm với mẫu. Để cho chắc chắn, ta nên thực sự tiến hành so sánh trực tiếp văn bản đó với mẫu. Tuy nhiên, việc sử dụng giá trị rất lớn của q, được biến thành dương bởi các phép toán mod và bởi sự kiện là ta không cần duy trì bảng băm thực sự, đã khiến cho rất khó xảy ra một sự đụng độ. Về mặt lý thuyết, thuật toán này có thể vẫn thực hiện theo O(NM) bước trong trường hợp xấu nhất (không đáng tin cậy), nhưng trong thực tế có thể dựa vào thuật toán để thực hiện khoảng N+M bước.

ĐA TRUY (Multiple Searches)

Tất cả các thuật toán đang bàn đều hướng tới một bài toán tìm chuỗi cụ thể : tìm sự xuất hiện của một mẫu cho trước trong một chuỗi văn bản cho trước. Nếu cùng một chuỗi văn bản là đối tượng của nhiều phép tìm kiếm mẫu, thì sẽ là khôn ngoan để thực hiện một xử lý nào đó trên chuỗi sao cho các phép tìm kiếm kế tiếp được hiệu quả.

Nếu có một số lượng lớn các phép tìm kiếm, thì bài toán tìm chuỗi có thể được xem như 1 trường hợp đặc biệt của bài toán tìm kiếm tổng quát mà ta đã nghiên cứu trong phần trước. Đơn giản là ta xem chuỗi văn bản như là N "khoá" gối nhau, khoá thứ i được định nghĩa sẽ là a[1..N], toàn thể chuỗi văn bản bắt đầu ở vị trí i. Dĩ nhiên, ta không thao tác trên chính các khoá đó mà trên những con trỏ tới chúng : khi cần so sánh các khoá i và j, ta thực hiện các phép so sánh "từng ký tự một" bắt đầu ở các vị trí i và j trong chuỗi văn bản (nếu ta dùng một ký tự "cầm canh" (sentinel) nằm ở cuối, lớn hơn tất cả các ký tự khác, thì một trong các khoá là luôn lớn hơn khoá còn lại). Sau đó có thể dùng trực tiếp các thuật toán băm, cây nhị phân, và các thuật toán khác trong phần trước. Trước tiên, một cấu trúc toàn cục được xây dựng từ chuỗi văn bản, và sau đó có thể thực hiện các phép tìm kiếm hiệu quả đối với các mẫu cụ thể.

Nhiều chi tiết cần thiết sẽ được "xem xét qua" trong việc áp dụng các thuật toán tìm kiếm vào việc tìm kiếm chuỗi theo phương pháp này; dự định của chúng ta là chỉ ra điều này như một chọn lựa phụ cho một vài ứng dụng tìm-kiếm-chuỗi. Những phương pháp khác nhau sẽ thích hợp trong các trường hợp khác nhau. Lấy ví dụ, nếu các phép tìm kiếm áp dụng cho các mẫu có cùng độ dài, thì một bảng băm được kiến tạo với một lần quét duy nhất, giống như trong phương pháp Rabin-Karp, sẽ cho thời gian tìm kiếm là không đổi tính theo trung bình. Mặt khác, nếu các mẫu có độ dài biến thiên, thì một trong các phương pháp dựa-vào-cây (tree-based) có thể thích hợp (Patricia thì đặc biệt thích hợp cho một ứng dụng như vậy).

Các biến thể khác trong bài toán có thể khiến cho nó trở nên khó hơn đáng kể và dẫn tới các phương pháp khác nhau rất nhiều, như ta sẽ thấy trong hai chương kế tiếp.

BÀI TẬP

1. Cài đặt một thuật toán đối-sánh-mẫu brute-force mà nó quét mẫu từ phải sang trái.

2. Hãy cho bảng next đối với thuật toán Knuth-Morris-Pratt cho mẫu AAAAAAAA.

3. Hãy cho bảng next đối với thuật toán Knuth-Morris-Pratt cho mẫu ABRACADABRA.

4. Phác họa một máy trạng thái hữu hạn có thể tìm mẫu ABRACADABRA.

5. Làm thế nào bạn tìm được một chuỗi gồm 50 khoảng trống kế nhau trong một tập tin văn bản ?

6. Hãy cung cấp bảng skip từ-phải-sang-trái để quét mẫu ABRACADABRA từ-phải-sang-trái.

7. Hãy nghĩ ra một thí dụ sao cho việc quét mẫu từ phải-sang-trái với chỉ một heuristic "không khớp" thực hiện tồi.

8. Bạn sửa đổi như thế nào thuật toán Rabin-Karp để tìm một mẫu cho trước với điều kiện bổ sung là ký tự nằm giữa là một "ký tự đại diện" (wild card : bất kỳ một ký tự văn bản nào đều có thể sánh được với nó).

9. Cài đặt một phiên bản của thuật toán Rabin-Karp để tìm các mẫu trong một văn bản 2-chiều. Giả định cả mẫu lẫn văn bản đều là các hình chữ nhật ký tự.

10. Viết các chương trình để tạo ra một chuỗi văn bản ngẫu nhiên 1000-bit, sau đó tìm tất cả các xuất hiện của k bit cuối cùng ở một nơi khác trong chuỗi, với k=5,10,15. (Các phương pháp khác nhau có thể là thích hợp đối với các giá trị k khác nhau).

20
ĐỐI SÁNH MẪU

Thường ta muốn tìm chuỗi với một thông tin ít hơn thông tin hoàn chỉnh về mẫu sẽ được tìm. Ví dụ, những người sử dụng của một trình soạn thảo văn bản có thể chỉ muốn đưa ra một phần của một mẫu, hay chỉ ra một mẫu có khả năng trùng khớp ở một số ít từ khác nhau, hay chỉ ra rằng bất kỳ một số thể hiện nào bao gồm một vài ký tự đặc biệt sẽ được bỏ qua. Trong chương này ta sẽ xem xét làm thế nào có thể thực hiện phép đối sánh mẫu kiểu này một cách hiệu quả.

Các thuật toán trong chương trước có một sự phụ thuộc cơ bản vào đặc tả hoàn chỉnh của mẫu, vì vậy ta phải xem xét các phương pháp khác. Các cơ chế căn bản ta sẽ xem xét có khả năng tạo ra một phương tiện tìm chuỗi rất mạnh có khả năng đối sánh các mẫu M-ký tự phức tạp trong các chuỗi văn bản N-ký tự theo thời gian tỉ lệ với MN^2 trong trường hợp xấu nhất, và nhanh hơn nhiều đối với những ứng dụng đặc thù.

Đầu tiên, ta phải phát triển một phương pháp để mô tả các mẫu: một "ngôn ngữ" có thể được sử dụng để chỉ ra, theo một cách nghiêm ngặt, các kiểu bài toán tìm-chuỗi-riêng-phần được phát biểu ở trên. Ngôn ngữ này sẽ bao hàm các phép toán nguyên sơ (primitive operations), mạnh hơn phép toán đơn giản là "kiểm tra xem ký tự thứ i của chuỗi văn bản có khớp với ký tự thứ j của mẫu" được dùng trong chương trước. Trong chương này ta sẽ xem xét ba phép toán cơ bản theo cách nói của một kiểu máy tưởng tượng có thể tìm các mẫu trong một chuỗi văn bản. Thuật toán đối sánh mẫu của chúng ta sẽ là một phương pháp để giả lập phép toán của kiểu máy này. Trong chương kế tiếp ta sẽ thấy làm thế nào để dịch từ đặc tả mẫu mà người dùng khai thác, mô tả cho công việc tìm chuỗi của anh ta thành đặc tả máy mà thuật toán khai thác để thực thi việc tìm kiếm.

Như sẽ thấy, lời giải mà ta phát triển đối với bài toán đối sánh mẫu có liên hệ một cách mật thiết với các tiến trình cơ sở trong khoa học máy tính. Ví dụ, phương pháp ta sẽ dùng trong chương trình để thực hiện công việc tìm kiếm chuỗi suy từ một mô tả mẫu cho trước là cùng loại với phương pháp được dùng bởi hệ Pascal để thực hiện công việc tính toán có liên quan đến một chương trình Pascal cho trước.

MÔ TẢ MẪU

Ta sẽ xét các mô tả mẫu tạo ra bởi các ký hiệu, ràng buộc với nhau bởi ba phép toán cơ bản sau đây :

(i) Phép nối (concatenation) : Đây là thao tác được dùng trong chương trước. Nếu hai ký tự kề nhau trong mẫu, thì có một sự trùng khớp xảy ra nếu và chỉ nếu hai ký tự trong văn bản giống với nó cũng nằm kề nhau. Ví dụ, AB hiểu là A được theo sau bởi B.

(ii) Phép hội (or) : Đây là thao tác cho phép chúng ta chỉ ra các trường hợp trong mẫu. Nếu ta có một phép OR giữa hai ký tự, thì có một sự trùng khớp nếu và chỉ nếu một trong các ký tự đó xuất hiện trong văn bản. Ta sẽ ký hiệu thao tác này bằng cách dùng dấu + và dùng các dấu ngoặc đơn để tổ hợp nó với phép nối theo một cách thức phức tạp tùy ý. Ví dụ như, A+B nghĩa là "hoặc A hay B"; C(AC+B)D nghĩa là "hoặc CACD hoặc CBD"; và (A+C)((B+C)D) nghĩa là "hoặc ABD hoặc CBD hoặc ACD hoặc CCD".

(iii) Phép kết (closure) : Thao tác này cho phép các phần của một mẫu sẽ được lặp lại một cách tùy ý. Nếu ta có phép kết của một ký hiệu, thì sẽ có một sự trùng khớp xảy ra nếu ký hiệu đó xảy ra trong văn bản với số lần tùy ý (kể cả 0). Phép kết sẽ được ký hiệu bởi việc đặt một dấu * sau ký tự hay nhóm ký tự được bao bởi các dấu ngặc đơn sẽ được nhắc lại. Ví dụ, AB* đối sánh các chuỗi gồm có một chữ A theo sau là một dãy tùy ý các chữ B, trong khi (AB)* sẽ đối sánh các chuỗi gồm các chữ A và B khác nhau.

Một chuỗi các ký hiệu được xây dựng bằng cách dùng đến ba phép toán này được gọi là một biểu thức chính quy (regular expression). Mỗi một biểu thức chính quy có thể xác định các mẫu văn bản đặc thù. Mục đích của chúng ta là phát triển một thuật toán xác định xem có bất kỳ một mẫu nào được mô tả bởi một biểu thức chính quy cho trước xuất hiện trong một chuỗi văn bản cho trước hay không.

Ta sẽ tập trung vào phép nối, phép hội, và phép kết theo thứ tự để minh họa các nguyên lý cơ bản trong việc phát triển một thuật toán đối-sánh-mẫu biểu-thức-chính-quy. Các bổ sung khác được tạo chung trong các hệ thống thực sự để cho tiện. Ví dụ, -A có thể hiểu là "trùng khớp bất kỳ ký tự nào ngoại trừ A". Phép toán NOT này giống như một phép toán OR ám chỉ tất cả các ký tự ngoại trừ A nhưng dễ dùng hơn nhiều. Tương tự, "?" có thể hiểu là "trùng khớp bất kỳ ký tự nào". Lại một lần nữa, phép toán này hiển nhiên là cô đọng hơn nhiều so với một phép toán OR lớn. Những ví dụ khác về các ký hiệu bổ sung để tạo ra đặc tả của những mẫu lớn dễ hơn, đó là các ký hiệu để đối sánh nơi bắt đầu hay kết thúc của một dòng, bất kỳ ký tự nào hay con số nào,...

Các phép toán này có khả năng biểu đạt rất đáng kể. Ví dụ như mẫu mô tả ?*(ie+ei)?* sẽ sánh với tất cả các từ có ie hay ei; (1+01)*(0+1) mô tả tất cả các chuỗi số 0 và 1, không có hai số 0 kế nhau. Hiển nhiên là có nhiều mẫu mô tả khác nhau cùng mô tả cho một chuỗi : ta phải gắng xác định ra các mẫu mô tả ngắn gọn chỉ khi nào ta thử viết các thuật toán hiệu quả.

Thuật toán đối sánh mẫu ta sẽ xem xét có thể được xem như một sự khái quát hoá của phương pháp tìm kiếm chuỗi brute-force từ-trái-sang-phải. (phương pháp đầu tiên đã thấy trong chương 19). Thuật toán đi tìm chuỗi con trái nhất trong chuỗi văn bản mà nó khớp với mẫu mô tả bằng cách quét chuỗi văn bản từ trái sang phải, kiểm tra ở mỗi vị trí để xem khi nào thì có một chuỗi con bắt đầu ở vị trí đó khớp với mẫu mô tả.

CÁC MÁY ĐỐI SÁNH MẪU

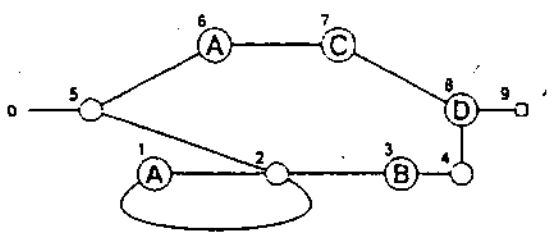
Nhớ lại là chúng ta có thể xem thuật toán Knuth-Morris-Pratt như

một máy trạng thái hữu hạn được kiến tạo từ mẫu tìm kiếm mà nó quét trên văn bản. Phương pháp ta sẽ dùng cho việc đối sánh mẫu biểu- thức-chính-quy là một sự khái quát hoá của thuật toán này.

Máy trạng-thái-hữu-hạn cho thuật toán Knuth-Morris-Pratt thay đổi từ trạng thái này sang trạng thái khác bằng cách nhìn vào một ký tự của chuỗi văn bản và sau đó chuyển đổi thành một trạng thái nào đó nếu có sự trùng khớp, nếu không thì thành một trạng thái khác. Một sự không trùng khớp ở bất kỳ một chỗ nào có nghĩa là mẫu không thể xảy ra trong văn bản bắt đầu ở điểm đó. Bản thân thuật toán có thể được xem như là sự mô phỏng cho một cái máy. Đặc trưng mà nó khiến cho dễ dàng mô phỏng máy là tính tất định (deterministic) : mỗi chuyển tiếp (transition) trạng thái hoàn toàn được xác định bởi ký tự nhập kế.

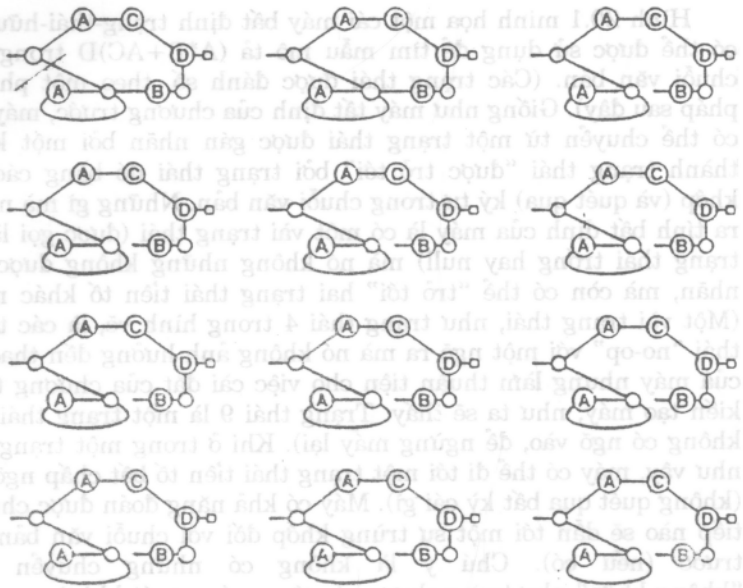
Để kiểm soát các biểu thức chính quy, cần xem xét một máy trừu tượng mạnh hơn. Do phép toán OR, máy không thể xác định khi nào thì mẫu có thể xuất hiện ở một vị trí cho trước bằng cách chỉ kiểm tra một ký tự : thực vậy, do phép kết, không thể xác định có bao nhiêu ký tự có thể cần phải được kiểm tra trước khi phát hiện ra một sự bất đối sánh. Cách tự nhiên nhất để vượt qua các vấn đề này là trang bị cho máy sức mạnh của tính bất định (nondeterminism) : khi có nhiều cách để đối sánh mẫu, máy sẽ "đoán ra" cái đúng ! Phép toán này có vẻ như không khả thi, nhưng ta sẽ thấy là dễ dàng viết một chương trình để mô phỏng các hành động của một cái máy như vậy.

Một máy nhận diện mẫu bất định cho (A*B+AC)D



Hình 20.1 minh họa một cái máy bất định trạng-thái-hữu-hạn có thể được sử dụng để tìm mẫu mô tả (A*B+AC)D trong một chuỗi văn bản. (Các trạng thái được đánh số, theo một phương pháp sau đây). Giống như máy tất định của chương trước, máy này có thể chuyển từ một trạng thái được gán nhãn bởi một ký tự thành trạng thái "được trỏ tới" bởi trạng thái đó bằng cách so khớp (và quét qua) ký tự trong chuỗi văn bản. Những gì mà nó tạo ra tính bất định của máy là có một vài trạng thái (được gọi là các trạng thái trống hay null) mà nó không những không được gán nhãn, mà còn có thể "trỏ tới" hai trạng thái tiền tố khác nhau. (Một vài trạng thái, như trạng thái 4 trong hình vẽ, là các trạng thái "no-op" với một ngõ ra mà nó không ảnh hưởng đến thao tác của máy nhưng làm thuận tiện cho việc cài đặt của chương trình kiến tạo máy, như ta sẽ thấy. Trạng thái 9 là một trạng thái null không có ngõ vào, để ngừng máy lại). Khi ở trong một trạng thái như vậy, máy có thể đi tới một trạng thái tiền tố bất chấp ngõ vào (không quét qua bất kỳ cái gì). Máy có khả năng đoán được chuyển tiếp nào sẽ dẫn tới một sự trùng khớp đối với chuỗi văn bản cho trước (nếu có). Chú ý là không có những chuyển tiếp "không-khớp" như trong chương trước : máy sơ sót khi tìm ra một sự trùng khớp chỉ khi không có cách, ngay cả để đoán ra một dãy các chuyển tiếp dẫn tới một sự trùng khớp.

Máy có một trạng thái khởi đầu duy nhất (được chỉ ra bởi đường không bị dính ở bên trái) và một trạng thái kết thúc duy nhất (hình vuông nhỏ ở bên phải). Khi được khởi động trong trạng thái khởi đầu, máy sẽ có khả năng "nhận diện" bất kỳ một chuỗi nào được mô tả bởi mẫu bằng cách đọc các ký tự và thay đổi trạng thái, tương ứng với các quy luật của nó, hoàn tất ở "trạng thái kết thúc". Do máy có khả năng bất định , nó có thể đoán ra dãy các thay đổi về trạng thái có thể dẫn tới lời giải. (Nhưng khi chúng ta thử mô phỏng máy trên một máy tính chuẩn, ta sẽ thử tất cả các khả năng). Ví dụ, để xác định xem mẫu mô tả (A*B+AC)D của nó có thể xảy ra trong chuỗi văn bản CDAABCAAABDDACDAAC hay không, máy sẽ ngay lập tức thông báo lỗi khi được khởi động trên ký tự thứ nhất hay thứ hai; nó sẽ làm một cái gì đó để thông báo lỗi trên các ký tự thứ năm và thứ sáu; và nó sẽ đoán ra dãy các chuyển tiếp trạng thái được chỉ ra trong hình 20.2 để nhận diện AAABD nếu được khởi động trên ký tự thứ bảy.

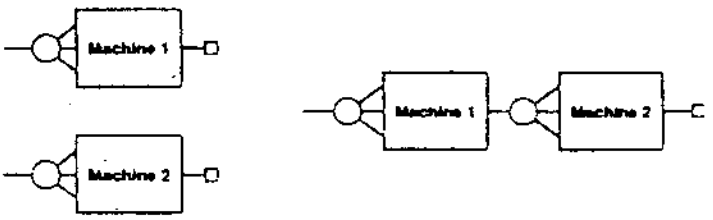


Hình 20.2 *Nhận dạng AAABD*

Chúng ta có thể kiến tạo một cái máy cho một biểu thức chính quy cho trước bằng cách xây dựng các máy bộ phận cho các phần của biểu thức và định nghĩa các phương pháp trong đó hai máy bộ phận có thể được tổ hợp lại thành một máy lớn hơn cho mỗi một phép toán trong số ba phép toán ở trên : phép nối, phép hội và phép kết.

Ta bắt đầu với một máy phụ để nhận diện một ký tự cụ thể. Thuận lợi khi viết nó như một máy hai trạng thái, với một trạng thái khởi đầu (mà nó cũng nhận diện ký tự) và một trạng thái kết thúc, như được minh họa trong hình 20.3.

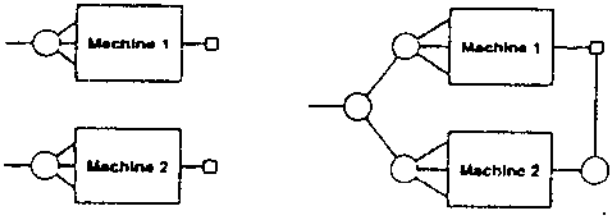
Hình 20.3 *Máy hai-trạng-thái để nhận diện 1 ký tự)*



Hình 20.4 *Kiến tạo máy trạng thái : phép nối*

Bây giờ để tạo cái máy cho phép nối của hai biểu thức từ các máy cho các biểu thức riêng lẻ, đơn giản là ta trộn trạng thái kết thúc của máy đầu với trang thái khởi đầu của máy thứ hai, như được minh họa trong hình 20.4.

Tương tự, máy cho phép toán hội OR được xây dựng bằng cách thêm một trạng thái null mới trỏ tới 2 trạng thái khởi đầu và tạo ra một trạng thái kết thúc trỏ tới cái còn lại, mà nó trở thành trạng thái kết thúc của máy được tổ hợp, như được minh họa trong hình 20.5.



Hình 20.5 *Kiến tạo máy trạng thái : phép hội*

Cuối cùng, máy cho phép toán kết được xây dựng bằng cách làm cho trạng thái kết thúc trở thành trạng thái khởi đầu và để cho nó trỏ ngược lại trạng thái khởi đầu cũ và một trạng thái kết thúc mới, như được minh họa trong hình 20.6.

Hình 20.6 *Kiến tạo máy trạng thái : phép kết*

Một máy có thể được xây dựng tương ứng với bất kỳ một biểu thức chính quy nào bằng cách áp dụng liên tiếp các quy tắc này. Các trạng thái cho máy ví dụ ở trên được đánh số theo thứ tự khởi tạo khi máy được xây dựng bằng cách dò mẫu từ trái sang phải, như thế việc kiến tạo của máy từ các quy tắc ở trên có thể được lần theo một cách dễ dàng. Chú ý rằng chúng ta có một máy phụ 2-trạng-thái cho mỗi ký tự trong biểu thức chính quy và mỗi dấu + hay * sẽ khiến cho một trạng thái được khởi tạo (phép nối khiến cho một trạng thái sẽ bị xoá), như thế số trạng thái chắc chắn sẽ ít hơn hai lần số ký tự trong biểu thức chính quy.

BIỂU DIỄN MÁY

Tất cả các máy bất định của chúng ta sẽ được kiến tạo bằng cách chỉ dùng ba quy tắc kết hợp đã được phác thảo ở trên, và ta có thể lợi dụng cấu trúc đơn giản của chúng để sử dụng chúng theo một cách thức đơn giản. Ví dụ, không có nhiều hơn hai đường rời khỏi bất kỳ một trạng thái nào. Thực vậy, chỉ có hai kiểu trạng thái : các trạng thái được gán nhãn bởi một ký tự từ bảng chữ cái nhập vào (với một đường đi ra), và các trạng thái không được gán nhãn (null) (với hai đường, hay ít hơn, rời khỏi nó). Điều này có nghĩa là máy có thể được biểu diễn chỉ với một vài mẩu thông tin trên một nút (node). Vì ta sẽ thường xuyên muốn truy xuất các trạng thái chỉ bằng con số, cách tổ chức phù hợp nhất cho máy sẽ là một biểu diễn theo mảng. Ta sẽ dùng ba mảng đồng thời là ch, next1, và next2, được chỉ mục bởi state để biểu diễn và truy xuất máy. Nó sẽ có khả năng để qua được với 2/3 lượng không gian này, vì mỗi trạng thái thực sự chỉ dùng hai mẩu thông tin có nghĩa, nhưng ta sẽ từ bỏ sự cải tiến này để cho dễ hiểu và cũng vì mẩu mô tả thường là không quá dài.

Máy ở trên có thể được biểu diễn như trong hình 20.7. Các đầu

state	0	1	2	3	4	5	6	7	8	9
ch[state]		A		B			A	C	D	
next1[state]	5	2	3	4	8	6	7	8	9	0
next2[state]	5	2	1	4	8	2	7	8	9	0

Hình 20.7 *Biểu diễn mảng cho máy ở hình 20.1*

vào được chỉ mục bởi state có thể được thông dịch như các lệnh với máy bất định có dạng "Nếu bạn đang ở vị trí state và bạn thấy ch[state] thì quét ký tự và đi tới trạng thái next1[state] (hay next2[state])". Trạng thái 9 là trạng thái kết thúc trong ví dụ này, và trạng thái 0 là một trạng thái khởi-đầu-giả mà các điểm vào Next của nó là số hiệu của trạng thái khởi đầu thực sự. (Chú ý biểu diễn đặc biệt được dùng cho các trạng thái trống null với 0 hay 1 ngõ ra).

Ta đã thấy làm thế nào để tạo nên máy từ mẫu mô tả biểu thức chính quy và làm thế nào các máy như vậy có thể được biểu diễn như các mảng. Tuy nhiên, viết một chương trình để làm công việc dịch từ một biểu thức chính quy thành một biểu diễn máy bất định tương ứng lại là một vấn đề hoàn toàn khác. Thực vậy, ngay cả việc viết một chương trình để xác định xem một biểu thức chính quy cho trước hợp lệ hay không là một thách đố đối với người không quen (uninitiated). Trong chương kế, ta sẽ nghiên cứu thao tác này, được gọi là phân tích văn phạm (parsing) một cách chi tiết hơn nhiều. Hiện tại, ta sẽ giả định là phép dịch này đã được thực hiện, sao cho có thể dùng được các mảng ch, next1, và next2 để biểu diễn một cái máy bất định cụ thể tương ứng với mẫu mô tả biểu thức chính quy đang quan tâm.

MÔ PHỎNG MÁY

Bước cuối cùng trong việc phát triển một thuật toán đối-sánh-mẫu biểu-thức-chính-quy là viết một chương trình mà nó mô phỏng theo một cách nào đó thao tác của một cái máy đối-sánh-mẫu bất định. Ý tưởng để viết một chương trình mà nó có thể "đoán ra" câu trả lời đúng có vẻ là khôi hài. Tuy nhiên trong trường hợp này nó dẫn tới một điều là chúng ta có thể lưu giữ dấu vết của tất cả các máy có thể có theo một phương pháp có hệ thống, sao cho ta thực sự tìm ra được đúng cái cần tìm.

Có một khả năng là phát triển một chương trình đệ quy bắt chước một cái máy bất định (nhưng thử tất cả các khả năng thay vì đoán ra cái đúng). Thay vì dùng cách tiếp cận này, ta sẽ xem thử một cài đặt không đệ quy biểu diễn các nguyên lý thao tác cơ bản của phương pháp bằng cách lưu các trạng thái qua việc xem

xét một cấu trúc dữ liệu đặc biệt khác được gọi là deque.

Ý tưởng là lưu giữ dấu vết của tất cả các trạng thái mà nó có thể bị bắt gặp trong khi máy đang "nhìn vào" ký tự nhập hiện thời. Mỗi một trạng thái được xử lý lần lượt : các trạng thái null dẫn tới 2 trạng thái (hay ít hơn), các trạng thái cho các ký tự mà nó không khớp với cái nhập vào hiện thời thì được loại ra, và những trạng thái cho các ký tự mà nó không khớp với cái nhập vào hiện thời sẽ dẫn tới các việc dùng các trạng thái mới khi máy đang nhìn vào ký tự nhập kế tiếp. Vì vậy ta muốn duy trì một danh sách của tất cả các trạng thái mà máy bất định có thể có ở một điểm cụ thể trong văn bản. Vấn đề là thiết kế một cấu trúc dữ liệu thích hợp cho danh sách này.

Việc xử lý các trạng thái null có vẻ cần một ngăn xếp (stack), vì chủ yếu ta đang trì hoãn việc thực hiện một trong hai điều, như trong việc khử đệ quy (như vậy trạng thái mới sẽ được đặt ở nơi bắt đầu của danh sách hiện hành, chỉ e rằng nó bị trì hoãn vô hạn). Việc xử lý các trạng thái khác có vẻ như cần đến một hàng đợi (queue), vì ta không muốn xác định các trạng thái cho ký tự nhập kế tiếp cho đến khi ta đã xong ký tự hiện hành (như vậy trạng thái mới sẽ được đặt ở cuối của danh sách hiện tại). Thay vì chọn một trong hai cấu trúc dữ liệu này, chúng ta sẽ dùng cả hai ! Deques (Các hàng hai đầu :"double-ended queues") là tổ hợp các đặc trưng của ngăn xếp và hàng đợi : một deque là một danh sách mà các phần tử có thể được thêm vào ở cả hai đầu. (Thực ra, ta dùng một "deque kết-xuất-hạn-chế (output-restricted queue)", vì ta luôn luôn bỏ đi các mẩu tin ở đầu, không phải ở cuối).

Một tính chất quyết định của máy là nó không có các "vòng lặp" mà chỉ gồm các trạng thái null, vì nếu không nó có thể quyết định một cách không xác định là lặp vĩnh viễn. Từ đó dẫn tới một điều là số trạng thái trên hàng đợi ở bất kỳ thời điểm nào là bé hơn số ký tự trong mẫu mô tả.

Chương trình được cho dưới đây dùng một hàng đợi hai đầu (deque) để mô phỏng các hành động của một máy đối-sánh-mẫu bất định như đã mô tả ở trên. Khi đang kiểm tra một ký tự cụ thể trong dãy nhập, máy bất định có thể ở bất kỳ một trạng thái nào trong số những trạng thái sau đây : chương trình lưu giữ dấu vết của những ký tự này trong một deque, bằng cách dùng các thủ tục push, put, và pop, giống như trong chương 3. Có thể dùng biểu diễn mảng (như trong việc cài đặt hàng đợi ở chương 3) hoặc một

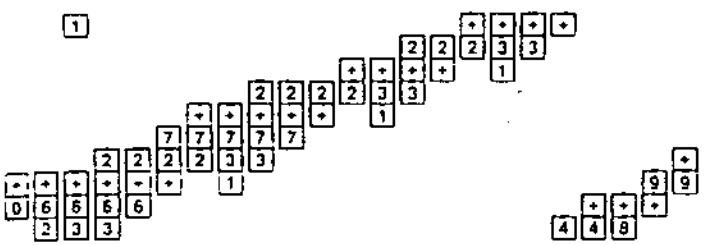
biểu diễn xâu liên kết (như trong việc cài đặt ngăn xếp ở chương 3); việc cài đặt được bỏ qua.

Vòng lặp chính trong chương trình loại bỏ một trạng thái từ hàng đợi hai đầu và thực hiện hành động yêu cầu. Nếu một ký tự sắp được đối sánh, thì ký tự nhập sẽ được kiểm tra xem có phải là ký tự được yêu cầu hay không; nếu đúng, thì sự chuyển tiếp trạng thái sẽ bị tác động bởi việc đặt trạng thái mới ở cuối của deque (sao cho tất cả các trạng thái có liên quan đến ký tự hiện thời sẽ được xử lý trước những trạng thái có liên quan đến ký tự kế tiếp). Nếu trạng thái là null, hai trạng thái có thể mà nó sẽ được mô phỏng sẽ được đặt ở nơi bắt đầu của deque. Các trạng thái có liên quan đến ký tự nhập hiện tại được lưu riêng khỏi các ký tự có liên quan đến văn bản bằng một dấu hiệu scan = -1 trong deque : khi scan bị bắt gặp, con trỏ trỏ vào trong chuỗi nhập sẽ được đẩy tới trước. Vòng lặp kết thúc khi tới cuối dãy nhập (không tìm thấy một sự trùng khớp nào), hoặc tới trạng thái 0 (đã tìm thấy sự trùng khớp hợp lệ), hoặc chỉ một phần tử, dấu hiệu scan, là còn lại trên deque (không tìm được sự trùng khớp). Điều này trực tiếp dẫn tới cài đặt sau đây :

```
function match(j:integer):integer;
  const scan=-1;
  var state,n1,n2:integer;
  begin
    dequeinit; put(scan);
    match:=j-1; state:=next1[0];
    repeat
      if state=scan then
        begin j:=j+1; put(scan) end
      else if ch[state]=a[j] then
        put(next1[state])
      else   if ch[state]='' then
        begin
          n1:=next1[state]; n2:=next2[state];
          push(n1);    if n1<>n2 then push(n2)
        end;
      state:=pop;
    until (j>N) or (state=0) or (dequeempty);
    if state=0 then match:=j-1;
  end;
```

Hàm này lấy tham số là vị trí j trong chuỗi văn bản a ở đó nó sẽ bắt đầu thử đối sánh. Nó trả về chỉ mục của ký tự cuối cùng khi đã tìm thấy sự trùng khớp (nếu có; nếu không nó trả về j-1).

Hình 20.8 chỉ ra nội dung của deque mỗi lần có một trạng thái được bỏ đi lúc máy mẫu của chúng ta thi hành với chuỗi văn bản là AAABD. Đồ hình này giả định một biểu diễn mảng, như đã được sử dụng cho các hàng đợi trong chương 3 : một dấu cộng được dùng để biểu diễn scan. Mỗi thời điểm dấu hiệu scan tìm tới đầu deque (ở trên đáy của đồ hình), thì con trỏ j được tăng lên tới ký tự kế trong văn bản. Vì vậy, ta bắt đầu với trạng thái 5 trong khi quét ký tự đầu tiên trong văn bản (chữ A đầu tiên). Trước tiên, trạng thái 5 dẫn tới trạng thái 2 và 6, sau đó trạng thái 2 dẫn tới các trạng thái 1 và 3, tất cả đều cần quét cùng một ký tự và ở nơi bắt đầu của deque. Sau đó trạng thái 1 dẫn tới trạng thái 2, nhưng ở cuối của deque (cho ký tự nhập kế tiếp). Trạng thái 3 dẫn tới trạng thái khác chỉ khi đang quét một chữ B, như thế nó được bỏ qua trong khi một chữ A đang được quét. Khi biến cầm canh "scan" tìm tới đầu của deque, ta thấy rằng máy có thể ở một trong hai trạng thái 2 hay 7 sau khi quét qua một A. Sau đó chương trình thử các trạng thái 2,1,3, và 7 trong khi "nhìn vào" chữ A thứ hai, để phát hiện ra là, trong lần thứ hai khi scan tìm tới đầu deque, trạng thái 2 là khả năng duy nhất sau khi quét AA. Bây giờ, trong khi nhìn vào ký tự A thứ ba, các khả năng duy nhất là các trạng thái 2, 1, và 3 (khả năng AC bây giờ bị ngăn chặn). Ba trạng thái này được thử lại lần nữa, để cuối cùng dẫn tới trạng thái 4 sau khi quét AAAB. Tiếp theo, chương trình chuyển sang trạng thái 8, dò chữ D và chuyển sang trạng thái kết thúc. Đã tìm thấy có sự trùng



Hình 20.8 *Nội dung của deque trong khi nhận diện AAABD*

khớp, nhưng, quan trọng hơn, là tất cả các chuyển tiếp mà nó nhất quán với chuỗi văn bản đều đã được xét đến.

TÍNH CHẤT 20.1 *Sự mô phỏng thao tác của 1 máy M-trạng-thái để tìm các mẫu trong một chuỗi văn bản gồm N ký tự ám chỉ NM chuyển tiếp trạng thái trong trường hợp xấu nhất.*

Thời gian chạy của chương trình này hiển nhiên phụ thuộc rất nhiều vào mẫu đang được so khớp. Tuy nhiên, đối với mỗi một ký tự trong số N ký tự nhập, nó xử lý hầu hết M trạng thái của máy, như thế thời gian thi hành trong trường hợp xấu nhất sẽ tỉ lệ với MN cho mỗi vị trí bắt đầu trong văn bản. Tổng thời gian để xác định xem có bất kỳ một phần nào trong chuỗi văn bản được mô tả bởi mẫu hay không, là $O(MN^2)$.

Không phải tất cả các máy bất định đều có thể được mô phỏng một cách hiệu quả như vậy, như sẽ được bàn luận chi tiết hơn trong chương 40, nhưng việc dùng một cái máy đối-sánh-mẫu giả thiết đơn giản trong ứng dụng này dẫn tới một thuật toán hoàn toàn hợp lý cho một bài toán thật khó. Tuy nhiên, để hoàn tất thuật giải, ta cần một chương trình dịch các biểu thức chính quy tùy ý thành các "cái máy" dùng cho việc thông dịch bởi chương trình ở trên. Trong chương kế tiếp, ta sẽ xem xét việc cài đặt của một chương trình như vậy trong ngữ cảnh một bàn luận khái quát hơn về các trình biên dịch và các kỹ thuật phân tích cú pháp.

BÀI TẬP

1. Cho một biểu thức chính quy để nhận ra tất cả các xuất hiện của bốn số 1 liên tiếp hay ít hơn trong một chuỗi nhị phân.
2. Phác họa máy đối-sánh-mẫu bất định cho mẫu mô tả (A+B)*+C.
3. Hãy cho các chuyển tiếp trạng thái máy của bạn do bài tập trước tạo ra để nhận diện ABBAC.
4. Giải thích cách làm thế nào sửa đổi lại máy bất định để điều khiển hàm NOT.
5. Giải thích cách làm thế nào sửa đổi lại máy bất định để kiểm soát các ký tự "không cần quan tâm" (don't care).
6. Có bao nhiêu mẫu khác nhau có thể được mô tả bởi một biểu thức chính quy với M toán tử OR và không có toán tử kết.
7. Sửa đổi match để kiểm soát các biểu thức chính quy với hàm NOT và các ký tự "không quan tâm".
8. Chỉ ra làm thế nào để kiến tạo một mẫu mô tả có độ dài M và một chuỗi văn bản có độ dài N mà đối với nó thời gian thực thi của match là càng lớn càng tốt.
9. Tại sao deque trong match chỉ được quyền chứa một biến cầm canh scan duy nhất.
10. Hãy xác định nội dung của deque mỗi khi một trạng thái được bỏ qua khi dùng match để mô phỏng máy mẫu trong văn bản với chuỗi văn bản là ACD.

21
PHÂN TÍCH CÂU

Nhiều thuật toán cơ sở đã được phát triển để nhận ra các chương trình máy tính hợp lệ và để phân rã chúng thành một dạng thích hợp cho những công việc xử lý khác nữa. Thao tác này, được gọi là phân tích câu, có ứng dụng vượt khỏi phạm vi của khoa học máy tính, vì nó có liên hệ trực tiếp với việc nghiên cứu cấu trúc ngôn ngữ nói chung. Ví dụ, phân tích câu đóng một vai trò quan trọng trong các hệ thống mà nó cố gắng để "hiểu được" các ngôn ngữ tự nhiên (ngôn ngữ của con người) và trong các hệ thống để dịch từ một ngôn ngữ thành một ngôn ngữ khác. Một trường hợp đáng chú ý đặc biệt là dịch từ một ngôn ngữ máy tính "cấp cao" như Pascal (thích hợp cho việc sử dụng của con người) thành một ngôn ngữ "cấp-thấp" như hợp ngữ (assembly) hay ngôn ngữ máy (thích hợp cho việc thi hành trên máy). Chương trình thực hiện công việc dịch như vậy được gọi là một trình biên dịch (compiler). Thực ra, ta đã gặp một phương pháp phân tích câu, ở chương 4 khi xây dựng cấu trúc cây để biểu diễn một biểu thức số học.

Hai cách tiếp cận tổng quát được sử dụng cho việc phân tích câu. Các phương pháp từ-trên-xuống (top-down) cho phép tìm một chương trình hợp lệ bằng cách trước tiên tìm các thành phần của một chương trình hợp lệ, sau đó tìm các thành phần của các thành phần này,... cho đến khi các mẩu thông tin đủ nhỏ để đối sánh với dữ liệu nhập một cách trực tiếp. Các phương pháp từ-dưới-lên (bottom-up) đặt các mẩu dữ liệu nhập lại với nhau theo một phương pháp có cấu trúc để tạo ra các mẩu ngày càng lớn hơn cho đến khi một chương trình hợp lệ được tạo ra. Thông thường, các phương pháp từ-trên-xuống là đệ quy, các phương pháp từ-dưới-lên là lặp; các phương pháp từ-trên-xuống được coi là dễ cài đặt hơn, các phương pháp từ-dưới-lên được coi là hiệu quả hơn. Phương pháp trong chương 4 là từ-dưới-lên; trong chương này ta nghiên cứu phương pháp từ-trên-xuống một cách chi tiết.

Một sự xem xét đầy đủ các kết quả trong việc xây dựng bộ phân tích câu và trình biên dịch rõ ràng vượt quá khuôn khổ của quyển sách này. Tuy nhiên, bằng cách xây dựng một "trình biên dịch" đơn giản để hoàn chỉnh thuật toán đối-sánh-mẫu của chương trước, chúng ta sẽ có khả năng xem xét một vài khái niệm cơ bản có liên quan. Đầu tiên ta sẽ xây dựng một bộ phân tích từ-trên-xuống cho một ngôn ngữ đơn giản để mô tả các biểu thức chính quy. Sau đó ta sẽ sửa đổi bộ phân tích để tạo ra một chương trình dịch các biểu thức chính quy thành các máy đối-sánh-mẫu dùng bởi thủ tục match của chương trước.

Dự định của chúng tôi trong chương này là đưa ra một vài cảm nghĩ về các nguyên lý cơ bản của việc phân tích và biên dịch trong khi cùng lúc phát triển một thuật toán đối-sánh-mẫu hữu dụng. Chắc chắn là ta không thể xem xét hết các kết quả có liên quan ở một mức độ sâu hơn cần thiết. Độc giả nên chú ý là những khó khăn tế nhị có thể xảy ra trong việc áp dụng cùng một cách tiếp cận cho các bài toán tương tự nhau, và việc kiến tạo trình biên dịch là một lĩnh vực hoàn toàn đã được phát triển với một phạm vi rộng lớn các phương pháp cao cấp khả dụng cho các ứng dụng nghiêm túc.

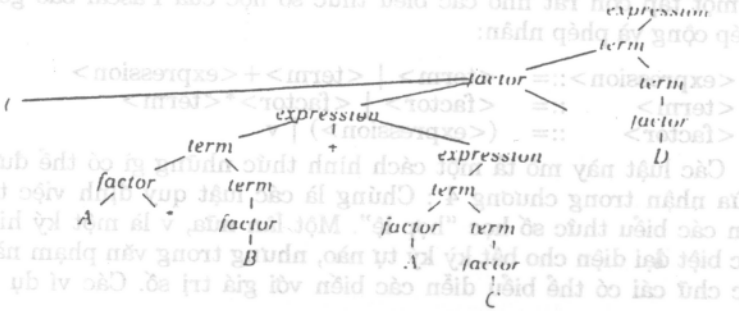
VĂN PHẠM PHI-NGỮ-CẢNH

Trước khi có thể viết được một chương trình để xác định xem một chương trình nào đó đã được viết trong một ngôn ngữ cho trước có là hợp lệ hay không, ta cần một mô tả chính xác về những gì cấu thành một chương trình hợp lệ. Mô tả này được gọi là một bộ văn phạm; để tán thành với thuật ngữ, hãy nghĩ đến một ngôn ngữ như tiếng Anh chẳng hạn và đọc "câu" cho "chương trình" trong câu trước đó (ngoại trừ lần xuất hiện đầu tiên !). Các ngôn ngữ lập trình thường được mô tả bởi một kiểu văn phạm cụ thể được gọi là văn phạm phi-ngữ-cảnh. Ví dụ, văn phạm phi ngữ cảnh định nghĩa tập tất cả các biểu thức chính quy (như đã mô tả trong chương trước) là như sau :

```
<expression>  ::=  <term> | <term> + <expression>
<term>        ::=  <factor> | <factor><term>
<factor>      ::=  (<expression>) | v | (<expression>) * | v *
```

Văn phạm này mô tả các biểu thức chính quy giống như đã được dùng trong chương trước, ví dụ như (1+01)*(0+1) hay (A*B+AC)D. Mỗi dòng trong văn phạm được được gọi là một luật sinh hay luật thay thế. Các luật sinh bao gồm các ký hiệu kết thúc (terminal) (,). + và * là những ký hiệu được dùng trong ngôn ngữ đang được mô tả ("v", một ký hiệu đặc biệt, đại diện cho bất kỳ một ký tự chữ hay số nào); các ký hiệu chưa kết (nonterminal) <expression>, <term> và <factor> là các ký hiệu trung gian của văn phạm; các siêu ký hiệu (meta symbols) ::= và | được dùng để mô tả ý nghĩa của các luật sinh. Ký hiệu ::=, đọc "là một", định nghĩa vế trái của luật sinh theo vế phải; và ký hiệu |, được đọc là "hoặc", chỉ ra các tùy chọn khác nhau. Các luật sinh khác nhau, dù được biểu diễn trong dạng ký hiệu ngắn gọn này, lại tương ứng một cách đơn giản với một mô tả trực quan của bộ văn phạm. Ví dụ, luật sinh thứ hai trong bộ văn phạm thí dụ có thể được đọc là "một <term> là một <factor> hoặc là một <factor> được theo sau bởi một <term>". Một ký hiệu chưa kết, trong trường hợp này được phân biệt theo ý nghĩa là một chuỗi các ký hiệu kết, là thuộc về ngôn ngữ đang được mô tả bởi bộ văn phạm nếu và chỉ nếu có một cách nào đó dùng các luật sinh để phát sinh ra chuỗi đó từ các ký hiệu chưa kết phân biệt bằng cách thay thế (theo số bước tùy ý) một ký hiệu chưa kết bằng bất kỳ một mệnh đề nào trong số các mệnh đề "hoặc" ở vế phải của một luật sinh cho ký hiệu chưa kết đó.

Một cách tự nhiên để mô tả kết quả của tiến trình phát sinh này là một cây phân tích : đó là một lược đồ cấu trúc văn phạm hoàn chỉnh của chuỗi đang được phân tích. Ví dụ, cây văn phạm ở



Hình 21.1 *Cây phân tích cho chuỗi (A*B+AC)D*

hình 21.1 cho thấy chuỗi (A*B+AC)D là nằm trong ngôn ngữ được mô tả bởi văn phạm ở trên. Các cây phân tích giống như cây này đôi khi được dùng cho tiếng Anh để ngắt một câu thành chủ ngữ, động từ, vị ngữ,...

Chức năng chính của một bộ phân tích là chấp nhận các chuỗi mà nó có thể được phát sinh như vậy và không nhận các chuỗi không thể được phát sinh như vậy, bằng cách thử tạo ra một cây phân tích cho bất kỳ một chuỗi nào. Nghĩa là, bộ phân tích có thể nhận diện khi nào thì một chuỗi là thuộc về ngôn ngữ đang được mô tả bởi văn phạm bằng cách xác định xem khi nào thì tồn tại một cây phân tích cho chuỗi đó. Các bộ phân tích từ-trên-xuống làm điều này bằng cách xây dựng cây bắt đầu bằng ký hiệu chưa kết nằm ở đỉnh và đi xuống dưới hướng về chuỗi sẽ được nhận diện nằm ở đáy; các bộ phân tích từ-dưới-lên thực hiện điều này bằng cách khởi đầu từ chuỗi nằm ở đáy và đi ngược lên trên hướng tới ký hiệu chưa kết phân biệt nằm ở đỉnh. Như ta sẽ thấy, nếu ý nghĩa các chuỗi đang được nhận dạng còn có liên quan đến việc xử lý khác nữa, thì bộ phân tích có thể chuyển chúng thành một biểu diễn bên trong mà nó có thể khiến cho việc xử lý như thế trở nên dễ dàng.

Một ví dụ khác của một bộ văn phạm phi-ngữ-cảnh có thể được tìm thấy trong Phụ lục của quyển sách "Pascal User Manual and Report" : quyển sách mô tả các chương trình Pascal hợp lệ. Các nguyên lý được xem xét trong phần này để nhận dạng và sử dụng các biểu thức hợp lệ được áp dụng trực tiếp trong việc biên dịch và thực hiện các chương trình Pascal. Ví dụ, bộ văn phạm sau đây mô tả một tập con rất nhỏ các biểu thức số học của Pascal bao gồm phép cộng và phép nhân:

```
<expression>::=  <term> | <term>+<expression>
<term>      ::=  <factor> | <factor>*<term>
<factor>    ::=  (<expression>) | v
```

Các luật này mô tả một cách hình thức những gì có thể được thừa nhận trong chương 4 : Chúng là các luật quy định việc tạo nên các biểu thức số học "hợp lệ". Một lần nữa, v là một ký hiệu đặc biệt đại diện cho bất kỳ ký tự nào, nhưng trong văn phạm này, các chữ cái có thể biểu diễn các biến với giá trị số. Các ví dụ về

những chuỗi hợp lệ cho văn phạm này là A+(B*C) và A*(((B+C)*(D*E))+F). Ta đã thấy cây phân tích cho chuỗi thứ hai ở trong chương 4, nhưng cây đó không ứng với văn phạm ở trên-Ví dụ, các dấu ngoặc đơn không được đề cập đến một cách tường minh.

Như đã xác định, một vài chuỗi là hợp lệ hoàn toàn cả về biểu thức số học lẫn biểu thức chính quy. Ví dụ, A*(B+C) có thể hiểu là "cộng B với C và nhân kết quả với A" hoặc "lấy một dãy tùy ý các chữ A theo sau bởi B hoặc C". Điều này cho thấy một sự kiện hiển nhiên là việc kiểm tra khi nào một chuỗi được tạo thành một cách hợp lệ là một chuyện, còn việc hiểu nó có ý nghĩa gì lại là một chuyện hoàn toàn khác. Ta sẽ trở lại vấn đề này sau khi đã xem làm thế nào phân tích một chuỗi để xác định xem nó được mô tả bởi một văn phạm nào đó hay không.

Mỗi một biểu thức chính quy bản thân nó là một ví dụ của một bộ văn phạm phi ngữ cảnh : bất kỳ một ngôn ngữ nào mà nó có thể được mô tả bởi một biểu thức chính quy thì cũng có thể được mô tả bởi một văn phạm phi ngữ cảnh. Điều ngược lại không đúng. Ví dụ, khái niệm về sự "cân bằng" số dấu ngoặc đơn không thể nắm bắt được bởi các biểu thức chính quy. Các kiểu văn phạm khác có thể mô tả các ngôn ngữ mà các văn phạm phi ngữ cảnh không thể mô tả được. Ví dụ, các văn phạm cảm ngữ cảnh (context-sensitive grammar) giống với văn phạm văn phạm phi ngữ cảnh, ngoại trừ vế trái của các luật sinh không nhất thiết là các ký hiệu chưa kết đơn lẻ. Những khác biệt giữa các lớp ngôn ngữ và công việc phân cấp các văn phạm để mô tả chúng đã được giải quyết một cách rất cẩn thận và tạo nên một lý thuyết đẹp đẽ, có vị trí trung tâm trong khoa học máy tính.

PHÂN TÍCH TỪ-TRÊN-XUỐNG

Một phương pháp phân tích sử dụng đệ quy để nhận dạng các chuỗi từ một ngôn ngữ được mô tả một cách chính xác như đã quy định bởi bộ văn phạm. Nói một cách đơn giản, văn phạm là một đặc tả của ngôn ngữ mà nó hoàn chỉnh đến nỗi nó có thể được chuyển trực tiếp thành chương trình !

Mỗi luật sinh ứng với một thủ tục có tên là ký hiệu chưa kết ở vế trái. Các ký hiệu chưa kết ở vế phải của dãy nhập tương ứng (có thể là đệ quy) với các lệnh gọi thủ tục; các ký hiệu kết tương ứng với việc dò trên chuỗi nhập vào. Ví dụ, thủ tục sau là một phần của một bộ phân tích từ-trên-xuống đối với bộ văn phạm biểu thức chính tắc của chúng ta :

```
procedure expression;
   begin
   term;
   if p[j]='+' then begin j:=j+1; expression end
   end;
```

Một mảng p chứa biểu thức chính quy đang được phân tích và một chỉ mục j trỏ tới ký tự hiện đang được kiểm tra. Để phân tích một biểu thức chính quy cho trước, ta đặt nó trong p[1..M] (với một ký tự cầm canh trong p[M+1] không được dùng trong văn phạm), đặt j là 1, và gọi expression. Nếu điều này khiến cho j được đặt về giá trị M+1, thì biểu thức chính quy là ở trong ngôn ngữ được mô tả bởi văn phạm. Nếu không, ta sẽ xem ở dưới đây các trường hợp sai khác nhau được kiểm soát như thế nào.

Điều đầu tiên mà expression làm là gọi term, được cài đặt hơi phức tạp hơn một chút :

```
procedure term;
   begin
   factor;
   if (p[j]='(') or letter(p[j]) then term
   end;
```

Một phép biên dịch trực tiếp từ văn phạm đơn giản là để cho term gọi factor và sau đó tới term. Điều này hiển nhiên sẽ không hoạt động được vì nó không có cách nào để thoát khỏi term : chương trình này sẽ đi vào một vòng lặp đệ quy vô tận nếu được gọi (các vòng lặp như vậy có những tác động thật không hay ho gì trong nhiều hệ thống). Việc cài đặt ở trên né qua được chỗ này bằng cách trước tiên kiểm tra dãy nhập vào để quyết định xem khi nào thì term sẽ được gọi. Việc đầu tiên mà term làm là gọi factor, là thủ tục duy nhất trong số các thủ tục có thể phát hiện ra một sự bất đối sánh trong dãy nhập. Từ bộ văn phạm, ta biết rằng khi

factor được gọi, thì ký tự nhập vào hiện thời phải là một ký tự "(" hoặc phải là một chữ cái nhập (biểu diễn bởi v). Tiến trình kiểm tra ký tự kế này mà không tăng j để quyết định xem sẽ làm gì thì được gọi là nhìn trước (lookahead). Đối với một vài bộ văn phạm, điều này là không cần thiết; đối với những bộ văn phạm khác, có thể cần nhiều lần nhìn trước hơn.

Giờ đây, cài đặt của factor tuân thủ một cách trực tiếp theo bộ văn phạm. Nếu ký tự nhập đang được quét không phải là một ký tự "(" hay một chữ cái nhập, thì một thủ tục error được gọi để kiểm soát trường hợp lỗi :

```
procedure factor;
   begin
   if p[j]='('
   then
      begin
      j:=j+1; expression;
      if p[j]=')' then j:=j+1 else error
      end
   else if letter(p[j]) then j:=j+1 else error;
   if p[j]='*' then j:=j+1
   end;
```

Một trường hợp sai khác xảy ra khi thiếu một dấu ")"

Các thủ tục expression, term và factor rõ ràng là đệ quy; thực vậy, chúng quyện vào nhau quá chặt đến nỗi ta không thể biên dịch được chúng trong Pascal nếu không dùng cấu trúc **forward** để né tránh quy tắc dựa trên việc dùng một thủ tục mà không khai báo chúng trước.

Cây phân tích đối với một chuỗi cho trước sẽ sinh ra cấu trúc gọi đệ quy trong lúc phân tích. Hình 21.2 dò theo thao tác của ba thủ tục ở trên khi p chứa (A*B+AC)D và expression được gọi với j=1. Ngoại trừ dấu cộng, toàn bộ việc "quét" là được thực hiện trong factor. Để dễ đọc, các ký tự mà thủ tục factor quét, ngoại trừ các dấu ngoặc, sẽ được đặt trên cùng một dòng với lệnh gọi factor.

Độc giả được khuyến khích để liên hệ tiến trình này với văn phạm và cây trong hình 21.1. Tiến trình này tương ứng với việc dịch chuyển trên cây theo tiền-thứ-tự, mặc dù sự tương ứng là

```
expression
  term
    factor
      (
      expression
        term
          factor   A
          term
            factor   B
        +
        expression
          term
            factor   A
            term
              factor   C
      )
  term
    factor   D
```

không chính xác do chiến lược nhìn trước của chúng ta chung quy là để thay đổi văn phạm. Vì ta bắt đầu ở đỉnh của cây và đi xuống, nguồn gốc của tên từ-trên-xuống là dễ hiểu. Các bộ văn phạm như thế cũng thường được gọi là các bộ phân tích đệ-quy-xuống do chúng di chuyển xuống phía dưới của cây phân tích một cách đệ quy.

Cách tiếp cận từ-trên-xuống sẽ không hoạt động được đối với tất cả các văn phạm phi ngữ cảnh có thể có. Ví dụ, với luật sinh <expression> ::= v | <expression> + <term>, nếu ta tuân theo cách dịch máy móc thành Pascal như ở trên, ta sẽ nhận được kết quả không mong muốn:

```
procedure badexpression;
  begin
    if letter(p[j]) then j:=j+1 else
      begin
        badexpression;
        if p[j]<>'+' then error else
          begin j:=j+1; term end
      end
  end;
```

Nếu thủ tục này được gọi với p[j] không phải là chữ cái (như trong ví dụ của ta, với j=1) nó sẽ đi vào một vòng lặp đệ quy vô hạn. Việc tránh những vòng lặp như vậy là một khó khăn chủ yếu

```
expression
   term
      factor
         (
         expression
            term
               factor    A
               term
                  factor    B
            +
            expression
               term
                  factor    A
                  term
                     factor C
         )
      term
         factor    D
```

Hình 21.2 *Phân tích (A*B+AC)D*

trong việc cài đặt các bộ phân tích đệ-quy-xuống. Đối với term, ta đã dùng việc nhìn trước để tránh một vòng lặp như vậy; trong trường hợp này cách thích hợp là đi đường vòng để tránh khó khăn này bằng cách chuyển văn phạm qua <term>+<expression>. Sự xuất hiện một ký hiệu chưa kết giống như cái đầu tiên trên vế phải của một luật thay thế cho chính nó thì được gọi là đệ-quy-trái. Thực ra, vấn đề là quái ác hơn, do việc đệ quy trái có thể xảy ra gián tiếp, ví dụ như với các luật sinh
<expression>::=<term> và
<expression>::=v|<expression>+<term>

Các bộ phân tích đệ quy xuống sẽ không hoạt động được đối với các văn phạm như vậy. Chúng phải được biến đổi thành các văn phạm tương đương mà không có sự đệ quy trái, hay phải dùng đến một phương pháp phân tích nào đó. Thông thường, có một mối liên lạc mật thiết và đã được nghiên cứu rất rộng rãi giữa các bộ phân tích và các bộ văn phạm mà chúng nhận dạng, và việc chọn lựa một kỹ thuật phân tích thường được truyền đạt bởi các đặc trưng của văn phạm sẽ phân tích.

PHÂN TÍCH TỪ-DƯỚI-LÊN

Mặc dù có nhiều lệnh gọi đệ quy trong các chương trình ở trên, có một bài tập hướng dẫn để loại đi sự đệ quy một cách có hệ thống. Nhớ lại ở chương 5 là mỗi lệnh gọi thủ tục có thể được thay bằng một thủ tục cất vào ngăn xếp và thay mỗi thủ tục trả về bởi một lệnh lấy khỏi ngăn xếp, bắt chước những gì hệ Pascal thực hiện để cài đặt sự đệ quy. Cũng vậy, nhớ lại rằng một lý do để làm điều này là nhiều lệnh gọi có vẻ như đệ quy nhưng lại không thực sự đệ quy. Khi một lệnh gọi thủ tục là hành động cuối cùng của một thủ tục, thì một lệnh goto đơn giản có thể được sử dụng. Nó chuyển expression và term vào những vòng lặp đơn giản có thể được trộn lại với nhau và được tổ hợp với factor để sinh ra một thủ tục duy nhất với một lệnh đệ quy thực sự (lệnh gọi tới expression trong factor).

Cách nhìn này trực tiếp dẫn tới một phương pháp thật đơn giản để kiểm tra khi nào các biểu thức chính quy là hợp lệ. Một khi tất cả các lệnh gọi thủ tục đã được loại bỏ, ta thấy rằng mỗi ký hiệu kết thúc sẽ chỉ được quét khi nó bị bắt gặp. Việc xử lý duy nhất được thực hiện thực sự là kiểm tra xem có một dấu ngoặc phải sánh được với mỗi một dấu ngoặc trái hay không, khi nào thì mỗi dấu "+" là được theo sau bởi một chữ cái hoặc một dấu "(", và khi nào thì mỗi dấu "*" là được theo sau bởi một chữ cái hoặc một dấu ")". Nghĩa là, việc kiểm tra khi nào một biểu thức chính quy là hợp lệ sẽ chủ yếu tương đương với việc kiểm tra xem có sự cân bằng về số dấu ngoặc hay không. Điều này có thể được cài đặt một cách đơn giản bằng cách duy trì một biến đếm (counter), được khởi động là 0, được tăng lên khi bắt gặp một dấu ngoặc trái và được giảm đi khi bắt gặp một dấu ngoặc phải. Nếu biến đếm là 0 ở cuối biểu thức, và các ký hiệu "+" và "*" trong biểu thức thoả các yêu cầu vừa được nêu, thì biểu thức là hợp lệ.

Dĩ nhiên, có những phân tích khác hơn là chỉ kiểm tra khi nào dãy nhập vào là hợp lệ : mục đích chính là để xây dựng cây phân tích (ngay cả theo cách ngầm định, như trong bộ phân tích từ-trên-xuống) cho công việc xử lý khác nữa. Có khuynh hướng là có thể làm điều này với các chương trình với cùng một cấu trúc chủ yếu như bộ kiểm tra dấu ngoặc đã được mô tả trong đoạn

trước. Một kiểu bộ phân tích mà nó thực hiện theo cách này thường được gọi là bộ phân tích giảm-phép-dịch-chuyển (shift-reduce parser). Ý tưởng là duy trì một ngăn xếp đẩy xuống (pushdown stack) chứa các ký hiệu kết và chưa kết. Mỗi bước trong việc phân tích sẽ hoặc là một bước dịch-chuyển, trong đó đơn giản là ký tự nhập kế sẽ được cất trên ngăn xếp, hoặc là một bước làm giảm, trong đó các ký tự tại đỉnh ngăn xếp được đối sánh với vế phải của một luật sinh nào đó trong văn phạm và được "rút lại thành" (được thay bằng) ký tự chưa kết trên vế trái của luật sinh đó. (Khó khăn chính trong việc xây dựng một bộ phân tích giảm-phép-dịch-chuyển là quyết định xem khi nào thì dịch chuyển và khi nào thì làm giảm. Đây có thể là một quyết định phức tạp, phụ thuộc vào văn phạm). Sau này, tất cả các ký tự nhập vào được dịch chuyển vào trong ngăn xếp, và cuối cùng ngăn xếp được làm giảm thành một ký hiệu chưa kết đơn lẻ. Các chương trình trong chương 3 và 4 dùng cho việc kiến tạo một cây phân tích từ một biểu thức infix bằng cách trước tiên chuyển biểu thức thành postfix là một ví dụ đơn giản về một bộ phân tích như vậy.

Phân tích từ-dưới-lên thường được xem là phương pháp lựa chọn cho các ngôn ngữ lập trình thực sự, và có một tài liệu mở rộng việc phát triển các bộ phân tích cho các bộ văn phạm lớn có kiểu được yêu cầu để mô tả một ngôn ngữ lập trình. Mô tả ngắn gọn của chúng ta chỉ lướt qua bề mặt của các kết quả đã được tạo ra.

TRÌNH BIÊN DỊCH

Một trình biên dịch có thể được xem như một chương trình dịch một ngôn ngữ thành một ngôn ngữ khác. Ví dụ, một trình biên dịch Pascal dịch các chương trình từ ngôn ngữ Pascal thành ngôn ngữ máy của một máy cụ thể nào đó. Ta sẽ minh họa một cách để làm điều này bằng cách tiếp tục với thí dụ đối-sánh-mẫu biểu-thức-chính-quy; tuy nhiên bây giờ ta mong muốn dịch từ ngôn ngữ của các biểu thức chính quy thành một "ngôn ngữ" cho các máy đối-sánh-mẫu, các mảng ch, next1, và next2 của chương trình match ở chương trước.

Tiến trình dịch chủ yếu là "một-một" : Với mỗi ký tự trong mẫu (ngoại trừ các dấu ngoặc đơn), ta muốn sinh ra một trạng thái cho máy đối-sánh-mẫu (một đầu vào trong mỗi một mảng). Cái mẹo là lưu giữ dấu vết của thông tin cần thiết để đổ vào trong các mảng next1 và next2. Để làm điều đó, ta sẽ chuyển mỗi một thủ tục trong bộ phân tích đệ-quy-xuống thành các hàm mà nó khởi tạo các máy đối-sánh-mẫu. Mỗi hàm sẽ thêm vào các trạng thái mới khi cần vào cuối của các mảng ch, next1 và next2, và trả về chỉ mục của trạng thái khởi đầu của máy đã được khởi tạo (trạng thái cuối sẽ luôn luôn là đầu vào cuối cùng trong các mảng). Ví dụ, hàm được cho dưới đây đối với luật sinh của <expression> sẽ tạo ra các trạng thái "hoặc" (or) cho máy đối-sánh-mẫu :

```
function expression:integer;
  var t1,t2:integer;
  begin
  t1:=term; expression:=t1;
  if p[j]='+' then
     begin
     j:=j+1; state:=state+1; t2:=state;
     expression:=t2; state:=state+1;
     setstate(t2,'',expression,t1);
     setstate(t2-1,'',state,state);
     end;
  end;
```

Hàm này sử dụng một thủ tục setstate mà nó chỉ đơn giản là đặt các đầu vào cho các mảng ch, next1 và next2 được chỉ mục bởi tham số đầu tiên chứa các giá trị được cho trong các tham số thứ hai, thứ ba và thứ tư, tương ứng. Chỉ mục state lưu giữ tình trạng "hiện hành" của máy đang được xây dựng : Mỗi khi một trạng thái mới được tạo ra, state được tăng lên. Vì vậy các chỉ mục trạng thái cho máy sẽ tương ứng với một dãy lệnh gọi thủ tục cụ thể giữa giá trị của state ở đầu vào và giá trị của state ở đầu ra. Chỉ mục trạng thái cuối là giá trị của state lúc thoát ra (chúng ta không thực sự "tạo" các trạng thái cuối cùng bằng cách tăng state trước khi thoát ra, vì điều này khiến cho nó dễ "trộn lẫn" trạng thái cuối cùng với các trạng thái khởi đầu sau đó, như ta sẽ thấy dưới đây).

Với quy ước này, ta dễ kiểm chứng (chú ý lệnh gọi đệ quy !) là

chương trình ở trên cài đặt quy tắc cho việc tổ hợp hai máy với phép toán or như đã phác họa trong chương trước. Trước hết máy dành cho phần đầu của biểu thức được xây dựng (một cách đệ quy), sau đó hai trạng thái null mới được thêm vào và phần thứ hai của biểu thức được xây dựng. Trạng thái null đầu tiên (với chỉ mục t2-1) là trạng thái cuối của máy cho phần đầu của biểu thức mà nó được chế tạo thành một trạng thái "no-op" để nhảy tới trạng thái kết thúc của máy cho phần thứ hai của biểu thức, như được yêu cầu. Trạng thái null thứ hai (với chỉ mục t2) là trạng thái khởi đầu, như thế chỉ mục của nó là giá trị trả về cho expression và các đầu vào next1 và next2 được tạo ra để chỉ tới những trạng thái khởi đầu của hai biểu thức. Chú ý là những biểu thức này được cấu tạo theo thứ tự đảo ngược với những gì ta mong muốn, do giá trị của state cho trạng thái no-op là không biết được cho đến khi lệnh gọi đệ quy tới expression được tạo ra.

Hàm cho <term> trước tiên xây dựng một cái máy cho một <factor> và sau đó, nếu cần, trộn trạng thái kết thúc của máy đó với trạng thái khởi đầu của máy cho một <factor> khác. Điều này dễ dàng được thực hiện hơn là như đã nói, vì state là chỉ mục trạng thái kết thúc của lệnh gọi tới factor. Một lệnh gọi tới term mà không tăng state thực hiện mẹo sau :

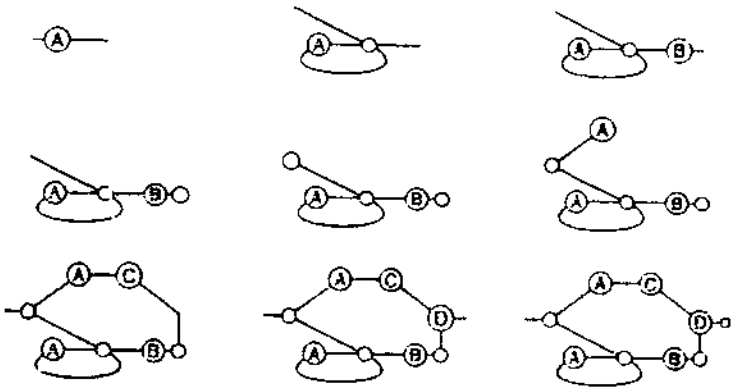
```
function term;
   var t:integer;
   begin
   term:=factor;
   if (p[j]='(') or letter(p[j]) then t:=term
   end;
```

(Chúng ta không cần dùng đến chỉ mục trạng thái khởi đầu được trả về bởi lệnh gọi thứ hai tới term, nhưng Pascal yêu cầu chúng ta đặt nó ở đâu đó, do đó ta sẽ bỏ nó vào một biến tạm t).

Hàm cho <factor> sử dụng các kỹ thuật tương tự để kiểm soát ba trường hợp của nó : một dấu ngoặc đại diện cho một lệnh gọi đệ quy trên expression; một ký hiệu v đại diện cho phép nối một trạng thái mới cuối chuỗi; và một dấu * đại diện cho các phép toán tương tự với các phép toán trong expression, tương ứng với lược đồ kết (closure diagram) của phần trước :

Hình 21.3 cho thấy làm thế nào các trạng thái được kiến tạo

```
function factor;
   var t1,t2:integer;
   begin
   t1:=state;
   if p[j]='(' then
      begin
         j:=j+1; t2:=expression;
         if p[j]=')' then j:=j+1 else error
      end
   else if letter(p[j]) then
      begin
         setstate(state,p[j],state+1,state+1);
         t2:=state; j:=j+1; state:=state+1
      end
   else error;
   if p[j]<>'*' then factor:=t2 else
      begin
         setstate(state,'',state+1,t2);
         factor:=state; next1[t1-1]:=state;
         j:=j+1; state:=state+1;
      end;
   end;
```



Hình 21.3 *Xây dựng một máy đối sánh mẫu cho (A*B+AC)D*

cho mẫu (A*B+AC)D, ví dụ trong chương trước của chúng ta. Đầu tiên, trạng thái 1 được xây dựng cho A. Sau đó trạng thái 2 được xây dựng cho toán hạng kết (closure) và trạng thái 3 được gắn vào cho B. Kế đó, dấu "+" bị bắt gặp và các trạng thái 4, 5 được xây dựng bởi expression, nhưng các vùng của chúng không thể được đổ đầy cho đến khi thực hiện xong một lệnh gọi đệ quy tới expression, và điều này cuối cùng gây ra việc tạo dựng các trạng thái 6 và 7. Cuối cùng, phép nối của D được điều khiển bởi trạng thái 8, để lại trạng thái 9 như là trạng thái kết thúc.

Bước cuối cùng trong việc phát triển một thuật toán đối-sánh-mẫu biểu-thức-chính-quy là gắn các thủ tục này lại với nhau cùng thủ tục match :

```
procedure matchall;
  begin
    j:=1; state:=1; next[0]:=expression;
    setstate(state,'',0,0);
    for i:=1 to N-1 do if match(i)>=i then writeln(i);
  end;
```

Chương trình này in ra tất cả các vị trí ký tự trong một chuỗi văn bản a[1..N] ở đó có một mẫu p[1..M] dẫn tới sự trùng khớp.

CÁC TRÌNH BIÊN DỊCH CỦA TRÌNH BIÊN DỊCH (Compiler-Compilers)

Chương trình dùng cho việc đối-sánh-mẫu biểu-thức-chính-quy đã được phát triển trong chương này và chương trước là những chương trình hiệu quả và thật hữu ích. Một phiên bản của chương trình này cùng với một vài chi tiết tinh tế được thêm vào (để kiểm soát các ký tự "không cần quan tâm", ...) sẽ là một trong số những trình tiện ích được sử dụng nhiều nhất trên nhiều hệ thống máy tính.

Thật thú vị khi nhận thức thuật toán này từ một quan điểm triết học hơn. Trong chương này, ta đã xem xét các bộ phân tích dùng cho việc gỡ rối cấu trúc của các biểu thức chính quy, dựa trên một mô tả hình thức của các biểu thức chính quy bằng cách dùng một bộ văn phạm phi-ngữ-cảnh. Nói cách khác, ta đã dùng

văn phạm phi-ngữ-cảnh để chỉ ra một "mẫu" cụ thể : một dãy các ký tự với số dấu ngoặc là bằng nhau. Bộ phân tích sau đó kiểm tra xem mẫu có xuất hiện trong dãy nhập vào hay không (nhưng nó coi một sự đối sánh là hợp lệ chỉ khi nó phủ toàn bộ chuỗi nhập vào). Vì vậy các bộ phân tích, mà nó kiểm tra một chuỗi nhập xem có nằm trong tập các chuỗi được định nghĩa bởi một bộ văn phạm phi-ngữ-cảnh nào đó, và các bộ sánh mẫu, mà nó kiểm tra một chuỗi nhập xem có nằm trong tập các chuỗi được định nghĩa bởi một biểu thức chính quy nào đó, thì chủ yếu thực hiện cùng một chức năng ! Khác nhau chủ yếu là các bộ văn phạm phi-ngữ-cảnh có khả năng mô tả một lớp các chuỗi rộng hơn nhiều. Ví dụ, các biểu thức chính quy không thể mô tả tập tất cả các biểu thức chính quy.

Một điểm khác nhau nữa trong các chương trình là bộ văn phạm phi-ngữ-cảnh được "xây dựng vào trong" bộ phân tích, trong khi thủ tục match lại được "điều khiển bằng bảng" : cùng một chương trình làm việc với tất cả các biểu thức chính quy, một khi chúng đã được dịch thành dạng thích hợp. Hoá ra là có khả năng xây dựng các bộ phân tích được- điều-khiển-bằng-bảng theo cùng một cách, sao cho cùng chương trình có thể được dùng để phân tích tất cả các ngôn ngữ được mô tả bởi các văn phạm phi-ngữ-cảnh. Một trình phát sinh bộ phân tích (parser generator) là một chương trình lấy dữ liệu nhập là 1 bộ văn phạm và sinh ra kết xuất là một bộ phân tích cho ngôn ngữ được mô tả bởi văn phạm đó. Có thể thực hiện thêm một bước nữa : người ta có thể xây dựng nên các trình biên dịch được điều khiển bằng bảng cho cả hai ngôn ngữ nhập và xuất. Một trình-biên-dịch-của-trình-biên-dịch (compiler-compiler) là một chương trình có đầu vào là hai bộ văn phạm (và một đặc tả về các mối quan hệ giữa chúng) và sinh ra kết xuất là một trình biên dịch mà nó sẽ dịch các chuỗi từ một ngôn ngữ thành ngôn ngữ còn lại.

Các trình phát sinh bộ phân tích và các "trình biên dịch của trình biên dịch" là khả dụng cho những ứng dụng thông thường trong nhiều môi trường tính toán, và là những công cụ hoàn toàn hữu ích, có thể được dùng để sinh ra các bộ phân tích và các trình biên dịch hiệu quả và đáng tin cậy với một nỗ lực tương đối ít. Mặt khác, các bộ phân tích đệ-quy-xuống từ-trên-xuống có kiểu được

xem xét ở đây thì hoàn toàn thích dụng đối với những bộ văn phạm đơn giản phát sinh trong nhiều ứng dụng. Vì vậy, như với nhiều thuật toán đã xem xét, ta có một phương pháp đơn giản thích hợp cho các ứng dụng, và nhiều phương pháp cao cấp mà nó có thể dẫn tới việc nâng cao hiệu năng một cách đáng kể đối với các áp dụng quy-mô-lớn. Như đã phát biểu ở trên, chúng ta chỉ đi lướt qua bề mặt của môi trường đang được nghiên cứu rộng rãi này.

BÀI TẬP

1. Làm thế nào bộ phân-tích-đệ-quy-xuống tìm được một lỗi trong một biểu thức chính quy chưa hoàn chỉnh, ví dụ như (A+B)*BC+ ?

2. Hãy cho cây phân tích đối với biểu thức chính quy ((A+B)+(C+D)*)*.

3. Hãy mở rộng bộ văn phạm biểu thức số học để có thêm phép toán lũy thừa, div và mod.

4. Hãy cho một bộ văn phạm phi-ngữ-cảnh để mô tả tất cả các chuỗi không có hơn hai con số 1 nằm kế nhau.

5. Có bao nhiêu lệnh gọi thủ tục được dùng bởi bộ phân tích đệ-quy-xuống để nhận dạng một biểu thức chính quy theo số phép nối, phép hội, và phép kết và số dấu ngoặc ?

6. Hãy cho các mảng ch, next1 và next2 sinh ra từ việc xây dựng máy đối-sánh-mẫu cho mẫu ((A+B)+(C+D)*)*.

7. Sửa đổi văn phạm biểu thức chính quy để kiểm soát hàm "không" (not) và các ký tự "không cần quan tâm" (don't care).

8. Xây dựng một bộ đối sánh mẫu biểu-thức-chính-quy dựa trên bộ văn phạm cải tiến trong câu trả lời của chúng ta cho câu hỏi trên.

9. Hãy loại bỏ sự đệ quy khỏi trình biên dịch đệ-quy-xuống và đơn giản hoá chương trình kết quả càng nhiều càng tốt. So sánh thời gian chạy của các phương pháp không đệ quy và đệ quy.

10. Viết một trình biên dịch cho các biểu thức số học đơn giản được mô tả bởi bộ văn phạm trong tài liệu. Nó sẽ sinh ra một danh sách các "lệnh" cho một máy có ba thao tác : cất (push) giá trị của biến lên trên ngăn xếp; thêm (add) vào đỉnh ngăn xếp hai giá trị nằm trên ngăn xếp, loại bỏ chúng khỏi ngăn xếp, sau đó đặt kết quả ở đó; và nhân (multiply) hai giá trị trên đỉnh ngăn xếp, theo cùng cách thức.

22
NÉN TẬP TIN

Phần lớn, các thuật toán ta đã nghiên cứu chủ yếu được thiết kế sao cho chạy càng ít tốn thời gian càng tốt, kế đến mới là để tiết kiệm chỗ. Trong phần này, ta sẽ xem xét một vài thuật toán theo hướng ngược lại: các phương pháp được thiết kế chủ yếu để giảm bớt lượng không gian sử dụng mà không tốn quá nhiều thời gian chạy. Mỉa mai thay, các kỹ thuật ta sẽ xem xét để tiết kiệm chỗ lại là các phương pháp "mã hoá" từ lý thuyết thông tin mà nó đã được phát triển để tối thiểu hoá lượng thông tin trong các hệ truyền thông và vì vậy về gốc rễ là có dự định tiết kiệm về thời gian (chứ không phải về không gian).

Thông thường, hầu hết các tập tin máy tính có rất nhiều thông tin dư thừa. Các phương pháp ta sẽ xem xét làm tiết kiệm chỗ bằng cách khai thác một sự kiện là hầu hết các tập tin có một "nội dung thông tin" tương đối thấp. Các kỹ thuật nén tập tin thường được dùng cho các tập tin văn bản (trong đó có một số ký tự nào đó xuất hiện thường hơn nhiều so với các ký tự khác). Các tập tin "raster" mã hoá cho ảnh (mà nó có thể có những vùng lớn đồng nhất), và các tập tin dùng để biểu diễn âm thanh dưới dạng số hoá và các tín hiệu tương tự (analog signals) khác (các tín hiệu này có thể có các mẫu được lặp lại nhiều lần).

Ta sẽ xem một thuật toán cơ sở cho bài toán (mà nó vẫn rất hữu ích) và một phương pháp "tối ưu" cấp cao. Lượng không gian được tiết kiệm bởi các phương pháp này biến thiên tùy thuộc vào các đặc trưng của tập tin. Tiết kiệm khoảng từ 20% đến 50% là phổ biến đối với các tập văn bản, và tiết kiệm khoảng từ 50% đến 90% cho các tập tin nhị phân. Đối với một vài kiểu tập tin, ví dụ như các tập tin gồm có các bit ngẫu nhiên, thì lượng chỗ tiết kiệm được sẽ là ít. Thực vậy, thật thú vị khi để ý là bất kỳ một phương pháp nén đa năng nào cũng sẽ làm cho một vài tập tin trở nên dài

hơn (nếu không thì ta đã có thể áp dụng phương pháp đó một cách liên tục để sinh ra một tập tin nhỏ tùy ý).

Một mặt, có thể cãi lại rằng các kỹ thuật nén tập tin thì ít quan trọng hơn ta tưởng vì giá cả của các thiết bị lưu trữ trên máy tính đã giảm rất nhiều và người sử dụng có nhiều bộ nhớ hơn so với trước đây. Mặt khác, người ta cũng có thể cãi lại là các kỹ thuật nén tập quan trọng hơn ta tưởng bởi vì, do có quá nhiều bộ nhớ đang trong tình trạng được sử dụng, việc tiết kiệm do chúng tạo ra có thể là to lớn hơn. Các kỹ thuật nén cũng thích hợp cho các thiết bị lưu trữ cho phép truy xuất với tốc độ cao vượt bực và đương nhiên là tương đối đắt (và vì vậy nó nhỏ).

MÃ HÓA ĐỘ-DÀI-LOẠT (Run-length Encoding)

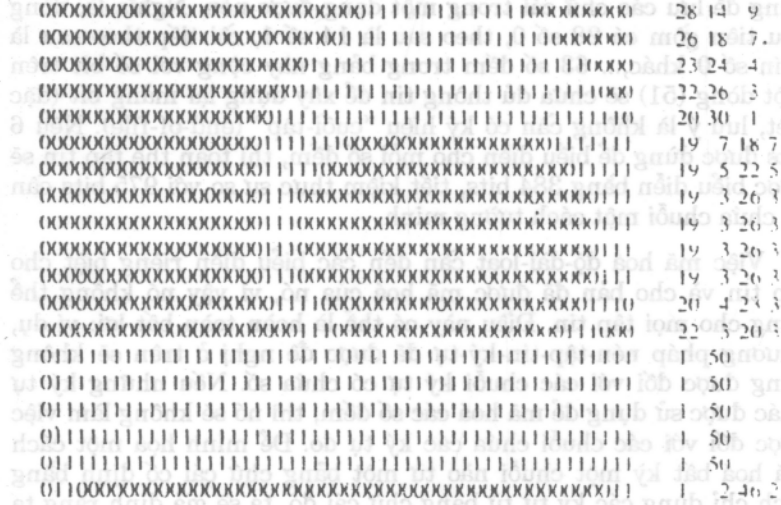
Loại dư thừa đơn giản nhất trong một tập tin là các đường chạy dài gồm các ký tự lặp lại. Ví dụ, xét chuỗi sau :

AAAABBBAABBBBBCCCCCCCCDABCBAAABBBBCCCD

Chuỗi này có thể được mã hoá một cách cô đọng hơn bằng cách thay thế từng chuỗi ký tự lặp lại bởi một thể hiện duy nhất của ký tự lặp lại cùng với một biến đếm số lần ký tự đó được lặp lại. Ta muốn nói rằng chuỗi này gồm bốn chữ A theo sau bởi ba chữ B rồi lại theo sau bởi hai chữ A, rồi lại theo sau bởi năm chữ B,... Việc nén một chuỗi theo phương pháp này được gọi là mã độ-dài-loạt. Khi có những loạt chạy dài, việc tiết kiệm có thể là đáng kể. Có nhiều cách để thực hiện ý tưởng này, tùy thuộc vào các đặc trưng của ứng dụng. (Các loạt chạy có khuynh hướng tương đối dài hay không ? Có bao nhiêu bit được dùng để mã hoá các ký tự đang được mã ?). Ta sẽ xem một phương pháp cụ thể, sau đó bàn luận đến các chọn lựa khác.

Nếu ta biết rằng chuỗi của chúng ta chỉ chứa các chữ cái, thì ta có thể mã hoá các biến đếm một cách đơn giản bằng cách xen kẽ các con số với các chữ cái. Vì vậy chuỗi của chúng ta có thể được mã hoá như sau :

4A3BAA5B8CDABCB3A4B3CD



Hình 22.1

Một bitmap cụ thể, với thông tin dùng cho việc mã hoá độ-dài-loạt

Ở đây "4A" có nghĩa là "bốn chữ A", ... Chú ý là không đáng để mã hoá các loạt chạy có độ dài 1 hoặc 2, vì cần đến hai ký tự để mã hoá.

Đối với các tập nhị phân, một phiên bản được tinh chế của phương pháp này được dùng để thu được sự tiết kiệm rất đáng kể. Ý tưởng đơn giản là lưu lại các độ dài loạt, tận dụng sự kiện các loạt chạy thay đổi giữa 0 và 1 để tránh phải lưu chính các số 0 và 1 đó. Điều này giả định rằng có một vài loạt chạy ngắn (ta tiết kiệm các bit trên một loạt chạy chỉ khi độ dài của đường chạy là lớn hơn số bit cần để biểu diễn chính nó trong dạng nhị phân), nhưng không có phương pháp mã hoá độ-dài-loạt nào hoạt động thật tốt trừ phi hầu hết các loạt chạy đều là dài.

Hình 22.1 là một biểu diễn "raster" của chữ cái "q" nằm trên cạnh đáy của nó; đây là biểu diễn của kiểu thông tin mà nó có thể được xử lý bởi một hệ định dạng văn bản (ví dụ như hệ được dùng

để in quyển sách này); ở bên tay phải là một danh sách các số được dùng để lưu các chữ cái trong một dạng được nén. Nghĩa là, dòng đầu tiên gồm có 28 số 0, theo sau là 14 số 1, rồi tiếp theo sau là chín số 0 khác,... 63 số đếm trong bảng này cộng với số bit trên một dòng (51) sẽ chứa đủ thông tin để xây dựng lại mảng bit (đặc biệt, lưu ý là không cần có ký hiệu "cuối-tập" (end-of-file). Nếu 6 bits được dùng để biểu diễn cho mỗi số đếm, thì toàn thể tập tin sẽ được biểu diễn bằng 384 bits, tiết kiệm thực sự so với 975 bits cần để chứa chuỗi một cách tường minh.

Việc mã hoá độ-dài-loạt cần đến các biểu diễn riêng biệt cho tập tin và cho bản đã được mã hoá của nó, vì vậy nó không thể dùng cho mọi tập tin. Điều này có thể là hoàn toàn bất lợi: ví dụ, phương pháp nén-tập-tin-ký-tự đã được đề nghị ở trên sẽ không dùng được đối với các chuỗi ký tự có chứa số. Nếu những ký tự khác được sử dụng để mã hoá các số đếm, thì nó sẽ không làm việc được đối với các chuỗi chứa các ký tự đó. Để minh họa một cách mã hoá bất kỳ một chuỗi nào từ một bảng chữ cái cố định bằng cách chỉ dùng các ký tự từ bảng chữ cái đó, ta sẽ giả định rằng ta chỉ có 26 chữ cái trong bảng chữ cái (và các khoảng trống) để làm việc.

Làm thế nào ta có thể tạo ra một vài chữ cái biểu diễn các số và các ký tự khác biểu diễn các phần của chuỗi sẽ được mã hoá ? Có một lời giải là dùng một ký tự được gọi là một ký tự "escape". Mỗi một sự xuất hiện của ký tự đó báo hiệu rằng hai chữ cái kế tiếp sẽ tạo thành một cặp (số đếm, ký tự), với các số đếm được biểu diễn bằng cách dùng ký tự thứ i của bảng chữ cái để biểu diễn số i. Vì vậy, chuỗi ví dụ của chúng ta sẽ được biểu diễn như sau với Q được xem như ký tự escape:

QDABBBAAQEBQHCDABCBAAAQDBCCCD

Tổ hợp của ký tự escape, số đếm, và một ký tự lặp lại thì được gọi là một dãy escape. Chú ý rằng không đáng để mã hoá các đường chạy có chiều dài ít hơn bốn ký tự, vì ít nhất là cần đến ba ký tự để mã hoá bất kỳ loạt chạy nào.

Nhưng điều gì sẽ xảy ra nếu bản thân ký tự escape xuất hiện trong dãy nhập vào ? Ta không thể cung cấp một cách đơn giản để

bỏ qua khả năng này, vì khó bảo đảm rằng bất kỳ một ký tự cụ thể nào là không thể xảy ra. (Ví dụ, một ai đó có thể thử mã hoá một chuỗi mà nó đã được mã hoá rồi). Một lời giải cho bài toán này là dùng một dãy escape với số đếm là 0 để biểu diễn ký tự escape. Vì vậy, trong ví dụ của chúng ta, ký tự khoảng trống (space) có thể biểu diễn số 0, và dãy escape "<Khoảng trống>" sẽ biểu diễn bất kỳ một sự xuất hiện nào của Q trong dãy đưa vào. Thật thú vị khi để ý rằng các tập tin chứa Q là các tập duy nhất được tạo ra dài hơn bởi phương pháp nén này. Nếu một tập tin đã được nén trước rồi bây giờ được nén lại lần nữa, thì nó sẽ phình ra thêm tối thiểu là một số ký tự bằng với số dãy escape được dùng.

Các loạt chạy rất dài có thể được mã hoá với nhiều dãy escape. Ví dụ, một loạt chạy gồm 51 chữ A sẽ được mã hoá như QZAQYA bằng cách dùng quy ước trên. Nếu nhiều loạt chạy rất dài được mong muốn, sẽ bõ công khi dành ra nhiều hơn một ký tự để mã hoá cho các số đếm.

Trong thực hành, nên làm cho các chương trình nén lẫn chương trình bung có một cái gì đó nhạy bén với các lỗi sai. Điều này có thể được thực hiện bằng cách thêm một lượng nhỏ thông tin dư thừa vào trong tập tin đã được nén sao cho chương trình bung có thể chịu được một sự thay đổi phụ tình cờ xảy ra trên tập tin giữa lúc nén và bung. Ví dụ, có thể bõ công khi đặt các ký tự "cuối dòng" (end-of-line) vào bản đã được nén của ký tự "q" ở trên, sao cho chương trình bung có thể tự đồng bộ lại trong trường hợp có một lỗi sai.

Mã hoá độ-dài-loạt không có hiệu quả đặc biệt đối với các tập tin văn bản do ký tự duy nhất rất hay được lặp lại là khoảng trống, và có những phương pháp đơn giản hơn để mã hoá các khoảng trống được lặp lại này. (xưa kia nó đã được dùng với ưu điểm lớn để nén các tập tin văn bản được tạo ra bằng cách đọc vào trong punched-card deck, chắc chắn có chứa nhiều khoảng trống). Trong các hệ thống hiện đại, các chuỗi khoảng trống được lặp lại thì không bao giờ được lưu trữ : các chuỗi khoảng trống lặp lại ở nơi bắt đầu của các dòng được mã hoá như các ký tự "tab" và các khoảng trống ở cuối của các dòng sẽ được tránh bằng cách dùng các dấu hiệu "cuối-dòng". Một cài đặt cho việc mã hoá độ-dài-loạt

giống như cài đặt ở trên (nhưng được sửa đổi để kiểm soát tất cả các ký tự hiện có) sẽ chỉ tiết kiệm khoảng 4% khi được dùng trên tập văn bản cho chương này (và sự tiết kiệm này tất cả đều xuất phát từ thí dụ về chữ "q" !).

MÃ HÓA ĐỘ DÀI BIẾN ĐỘNG (Variable-Length Encoding)

Trong phần này ta sẽ xác định một kỹ thuật nén-tập có thể tiết kiệm một lượng chỗ đáng kể trong các tập tin văn bản (và nhiều loại tập tin khác). Ý tưởng là từ bỏ phương pháp lưu trữ mà các tập tin văn bản thường sử dụng : thay vì thường dùng 7 hay 8 bit cho mỗi ký tự, thì chỉ dùng một vài bit cho các ký tự xuất hiện thường xuyên và dùng nhiều bit hơn cho các ký tự hiếm khi xuất hiện.

Sẽ là thuận lợi để xác định xem mã đã được dùng như thế nào trên một ví dụ nhỏ trước khi xét xem nó được tạo ra như thế nào. Giả sử ta muốn mã hoá chuỗi "ABRACADABRA". Mã hoá nó trong dạng mã nhị phân cô đọng chuẩn của chúng ta với biểu diễn nhị phân 5-bit của i biểu diễn ký tự thứ i của bảng chữ cái (0 thay cho khoảng trống) sẽ cho ra dãy bit sau đây :

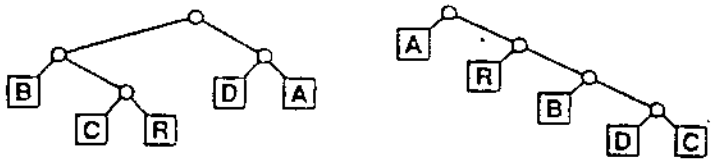
00001000101001000001000110000100100000010001010010000001

Để "giải mã" thông điệp này, đơn giản là đọc ra năm bits ở từng thời điểm và chuyển đổi nó tương ứng với việc mã hoá nhị phân đã được định nghĩa ở trên. Trong mã chuẩn này, chữ D xuất hiện chỉ một lần, sẽ cần một số lượng bit giống như A, xuất hiện năm lần. Với một mã độ-dài-biến-động, ta có thể đạt được sự tiết kiệm về chỗ bằng cách mã hoá các ký tự thường được dùng với càng ít bit càng tốt, sao cho tổng số các bit được dùng cho thông điệp là tối thiểu.

Ta có thể thử gán các chuỗi bit ngắn nhất cho các ký tự được dùng phổ biến nhất, mã hoá A là 0, B là 1, R là 01, C là 10, và D là 11, như thế ABRACADABRA sẽ được mã hoá thành

0 1 01 0 10 0 11 0 1 01 0

Ví dụ này chỉ dùng 15 bits so với 55 bits như ở trên, nhưng nó không thực sự là một mã vì nó lệ thuộc vào các khoảng trống để



Hình 22.2 *Hai tries mã hoá cho A,B,C,D và R*

phân cách các ký tự. Nếu không có khoảng trống, chuỗi 010101001101010 có thể được mã hoá như RRRARBRRA hay như nhiều chuỗi khác. Vẫn như vậy, số đếm 15 bits cộng thêm 10 ký tự phân cách thì cô đọng hơn là mã chuẩn, chủ yếu là do không có bit nào được dùng để mã hoá các chữ cái không xuất hiện trong thông điệp. Để cho công bằng, ta cũng cần đếm các bit trong chính mã đó, vì thông điệp không thể được mã hoá mà không có nó, và mã thì phụ thuộc vào thông điệp (các thông điệp khác sẽ có các tần số dùng các chữ khác nhau). Ta sẽ xem xét kết quả này sau; hiện tại ta đang quan tâm xem có thể cô đặc thông điệp ở mức độ nào.

Đầu tiên, các dấu phân cách là không cần thiết nếu không có mã ký tự nào là tiền tố (prefix) của một mã khác. Ví dụ, nếu ta mã hoá A là 11, B là 00, C là 010, D là 10, và R là 011, thì chỉ có một cách để giải mã chuỗi 25-bit :

1100011110101110110001111

Một cách dễ dàng để biểu diễn mã là bằng trie (xem Chương 17). Thực vậy, bất kỳ một trie nào với M nút ngoại có thể được dùng để mã hoá bất kỳ một thông điệp nào với M ký tự khác nhau. Ví dụ, Hình 22.2 cho thấy hai mã có thể được dùng cho ABRACADABRA. Mã cho mỗi ký tự được xác định bởi đường dẫn (path) từ gốc tới ký tự đó, với 0 dành cho "đi sang trái" và 1 dành cho "đi sang phải", giống như trong một trie. Vì vậy, trie ở bên trái ứng với mã được cho ở trên; trie ở bên phải ứng với mã sinh ra chuỗi

011010011110111001101O0

ngắn hơn hai bits. Việc biểu diễn theo trie bảo đảm rằng không có mã ký tự nào là tiền tố của một cái khác, như thế chuỗi là có thể được giải mã một cách duy nhất từ trie. Bắt đầu từ gốc, đi xuống trie tương ứng với các bit của thông điệp : mỗi khi một nút ngoại bị bắt gặp, ta xuất ra ký tự ở nút đó và bắt đầu lại ở gốc.

Nhưng dùng trie nào là tốt nhất ? Có một cách phù hợp để tính toán một trie mà nó dẫn tới một chuỗi bit có độ dài tối thiểu cho bất kỳ một thông điệp định trước nào. Phương pháp tổng quát để tìm ra mã đã được khám phá bởi D.Huffman vào năm 1952 và được gọi là mã hoá Huffman (Cài đặt mà ta sẽ xác định dùng một kỹ thuật thuật toán hiện đại hơn).

XÂY DỰNG MÃ HUFFMAN

Bước đầu tiên trong việc xây dựng mã Huffman là đếm tần số của mỗi ký tự trong thông điệp sẽ được mã hoá. Chương trình sau đây đổ vào một mảng count[0..26] các số đếm tần số cho một thông điệp trong một mảng ký tự a[1..M]. (Chương trình này dùng một thủ tục index đã được mô tả trong chương 19 để giữ số đếm tần số cho ký tự thứ i của bảng mẫu tự trong count[i], với count[0] được dùng cho các khoảng trống).

```
for i:=0 to 26 do count[i]:=0;
for i:=1 to M do count[index(a[i])]:=count[index(a[i])]+1;
```

Ví dụ, giả sử ta muốn mã hoá chuỗi

A SIMPLE STRING TO BE ENCODED USING A MINIMAL NUMBER OF BITS.

Bảng số đếm sinh ra được minh họa trong Hình 22.3 : có 11 khoảng trống, ba chữ A, ba chữ B,...

Bước kế là xây dựng trie mã hoá từ dưới lên tương ứng với các tần số. Trong việc xây dựng trie, ta sẽ xem nó như một cây nhị phân với các tần số được chứa trong các nút : sau khi nó đã được xây dựng ta sẽ xem nó như một trie cho việc mã hoá, như ở trên. Đầu tiên một nút của cây được khởi tạo cho mỗi một tần số khác 0,

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
k	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26
count[k]	11	3	3	1	2	5	1	2	0	6	0	0	2	4	5	3	1	0	2	4	3	2	0	0	0	0	0

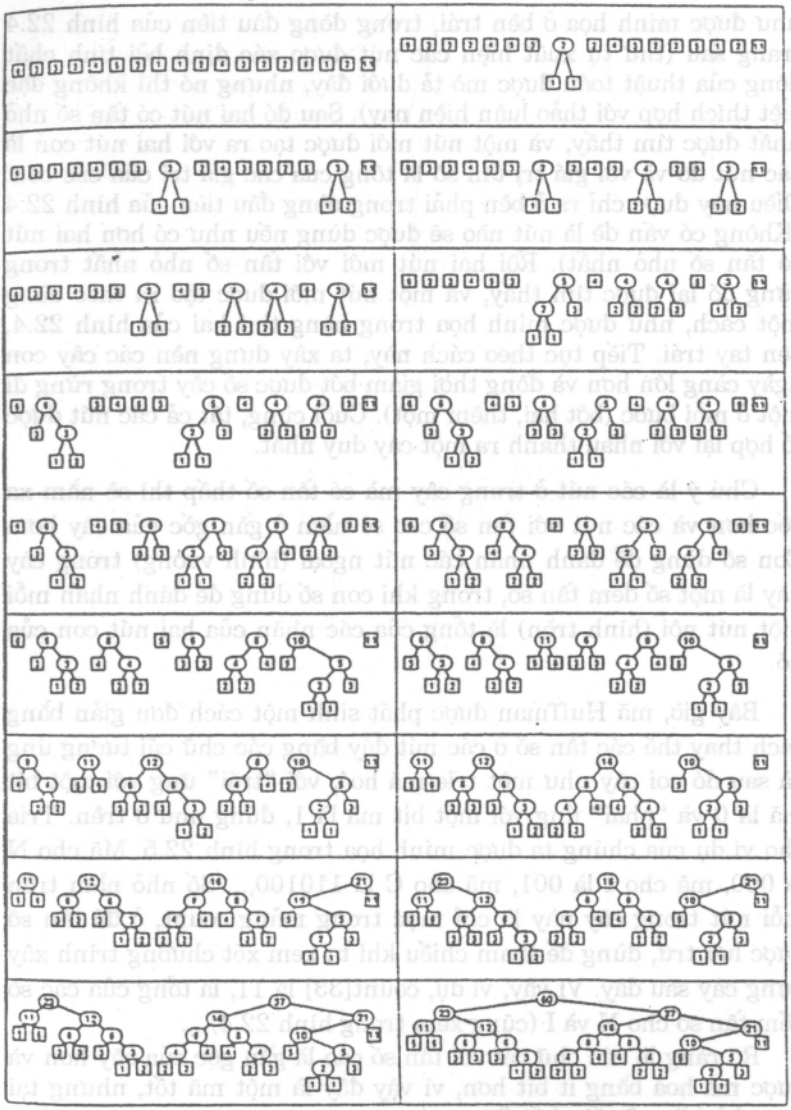
Hình 22.3 *Các số đếm tần số cho A SIMPLE STRING TO BE ENCODED . . .*

như được minh họa ở bên trái, trong dòng đầu tiên của hình 22.4 trang sau (thứ tự xuất hiện các nút được xác định bởi tính chất động của thuật toán được mô tả dưới đây, nhưng nó thì không đặc biệt thích hợp với thảo luận hiện nay). Sau đó hai nút có tần số nhỏ nhất được tìm thấy, và một nút mới được tạo ra với hai nút con là các nút đó và với giá trị tần số là tổng của các giá trị của các con. Điều này được chỉ ra ở bên phải trong dòng đầu tiên của hình 22.4 (Không có vấn đề là nút nào sẽ được dùng nếu như có hơn hai nút có tần số nhỏ nhất). Rồi hai nút mới với tần số nhỏ nhất trong rừng đó lại được tìm thấy, và một nút mới được tạo ra theo cùng một cách, như được minh họa trong hàng thứ hai của hình 22.4, bên tay trái. Tiếp tục theo cách này, ta xây dựng nên các cây con ngày càng lớn hơn và đồng thời giảm bớt được số cây trong rừng đi một ở mỗi bước (bớt hai, thêm một). Cuối cùng, tất cả các nút được tổ hợp lại với nhau thành ra một cây duy nhất.

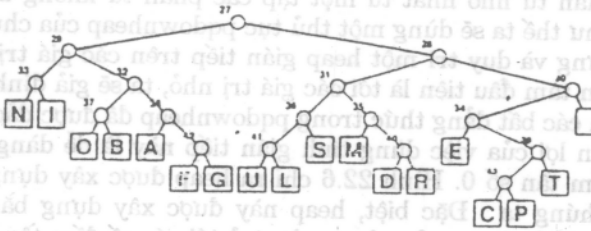
Chú ý là các nút ở trong cây mà có tần số thấp thì sẽ nằm xa gốc hơn và các nút với tần số cao sẽ nằm ở gần gốc của cây hơn. Con số dùng để đánh nhãn các nút ngoại (hình vuông) trong cây này là một số đếm tần số, trong khi con số dùng để đánh nhãn mỗi một nút nội (hình tròn) là tổng của các nhãn của hai nút con của nó.

Bây giờ, mã Huffman được phát sinh một cách đơn giản bằng cách thay thế các tần số ở các nút đáy bằng các chữ cái tương ứng và sau đó coi cây như một trie mã hoá, với "trái" ứng với một bit mã là 0 và "phải" ứng với một bit mã là 1, đúng như ở trên. Trie cho ví dụ của chúng ta được minh họa trong hình 22.5. Mã cho N là 000, mã cho I là 001, mã cho C là 110100,... Số nhỏ nằm trên mỗi nút trong cây này là chỉ mục trong mảng count, ở đó tần số được lưu trữ, dùng để tham chiếu khi ta xem xét chương trình xây dựng cây sau đây. Vì vậy, ví dụ, count[33] là 11, là tổng của các số đếm tần số cho N và I (cũng xem trong hình 22.4),...

Rõ ràng là các chữ cái với tần số cao là gần gốc của cây hơn và được mã hoá bằng ít bit hơn, vì vậy đây là một mã tốt, nhưng tại sao nó lại là mã tốt nhất ?



Hình 22.4 *Việc xây dựng một cây Huffman*



Hình 22.5 *Trie mã hoá Huffman cho câu A SIMPLE STRING TO BE ENCODED . . .*

TÍNH CHẤT 22.1 *Độ dài của thông điệp được mã hoá là bằng với độ dài đường dẫn ngoại được lấy trọng (weighted external path) của cây tần số Huffman.*

"Độ dài đường dẫn ngoại được lấy trọng" của một cây là tổng của tích tất cả các "trọng số" (số đếm tần số tương ứng) của các nút ngoại với khoảng cách tới gốc. Rõ ràng, đây là một cách để tính độ dài của thông điệp: Nó tương đương với tổng số lần xuất hiện trên tất cả các ký tự nhân với số bit của sự xuất hiện đó.

TÍNH CHẤT 22.2 *Không có cây nào với các tần số giống nhau trong các nút ngoại lại có đường dẫn ngoại được lấy trọng là nhỏ hơn cây Huffman.*

Bất kỳ một cây nào cũng có thể được cấu trúc lại bởi cùng một tiến trình mà ta dùng để xây dựng cây Huffman, nhưng không cần thiết phải nhặt ra hai nút có trọng nhỏ nhất ở mỗi bước. Có thể chứng minh quy nạp là không có chiến lược nào có thể làm tốt hơn chiến lược nhặt ra hai trọng số nhỏ nhất đầu tiên.

Mô tả ở trên cung cấp một cái sườn tổng quát cách làm thế nào để mã hoá Huffman, theo thuật ngữ của các thao tác thuật toán mà ta đã nghiên cứu. Như thông lệ, việc chuyển từ một mô tả tới một cài đặt thực sự là khá giáo điều, do đó ta sẽ xem xét các chi tiết của sự cài đặt ngay sau đây.

CÀI ĐẶT

Cấu trúc của cây tần số có liên quan đến tiến trình tổng quát để loại bỏ phần tử nhỏ nhất từ một tập các phần tử không được sắp thứ tự, như thế ta sẽ dùng một thủ tục pqdownheap của chương 11 để xây dựng và duy trì một heap gián tiếp trên các giá trị tần số. Vì ta quan tâm đầu tiên là tới các giá trị nhỏ, ta sẽ giả định rằng ý nghĩa của các bất đẳng thức trong pqdownheap đã được đảo ngược. Một thuận lợi của việc dùng tính gián tiếp này là dễ dàng bỏ qua các số đếm tần số 0. Hình 22.6 chỉ ra heap được xây dựng cho ví dụ của chúng ta : Đặc biệt, heap này được xây dựng bằng cách trước tiên khởi động mảng heap cho trỏ tới các số đếm tần số khác 0 và sau đó dùng thủ tục pqdownheap của chương 11, như sau :

```
N:=0;
for i:=0 to 26 do
    if count[i]<>0 then
        begin N:=N+1; heap[N]:=i end;
for k:=N downto 1 do pqdownheap(k);
```

Như đã đề cập ở trên, điều này giả định rằng ý nghĩa của các bất đẳng thức trong cài đặt của pqdownheap đã được đảo ngược.

Bây giờ, dùng thủ tục này để xây dựng cây như ở trên thì đơn giản: ta lấy hai phần tử nhỏ nhất ra khỏi heap, cộng chúng lại với nhau và đặt kết quả trở lại vào trong heap. Ở mỗi bước, ta khởi tạo một số đếm mới, và giảm bớt kích thước của heap đi một. Tiến trình này tạo ra N-1 số đếm mới, mỗi cái cho một trong các nút nội của cây đang được khởi tạo, như trong đoạn chương trình sau đây :

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
heap[k]	3	7	16	21	12	15	6	20	9	4	13	14	5	2	18	19	1	0
count[heap[k]]	1	2	1	2	2	3	1	3	6	2	4	5	5	3	2	4	3	11

Hình 22.6 *Heap khởi đầu (gián tiếp) cho việc tạo cây Huffman*

```
repeat
   t:=heap[1]; heap[1]:=heap[N]; N:=N-1;
   pqdownheap(1);
   count[26+N]:=count[heap[1]]+count[t];
   dad[t]:=26+N; dad[heap[1]]:=-26-N;
   heap[1]:=26+N; pqdownheap(1);
until N=1;
dad[26+N]:=0;
```

Hai dòng đầu tiên của vòng lặp này chính là pqremove; kích thước của heap được giảm đi một. Sau đó một nút nội mới được "khởi tạo" với chỉ mục 26+N và được cho một giá trị bằng với tổng của giá trị nằm ở gốc với giá trị vừa được loại ra. Sau đó mã này được đặt ở gốc, do đó tăng độ ưu tiên của nó lên, vì vậy cần đến một lệnh gọi khác tới pqdownheap để phục hồi lại trật tự trong heap. Bản thân cây được biểu diễn bởi một mảng các liên kết "cha con" : dad[t] là chỉ mục của cha của nút mà trọng số của nó ở trong count[t]. Dấu của dad[t] chỉ ra khi nào nút là con trái hay con phải của nút cha của nó. Ví dụ, trong cây ở trên ta có count[30]=21, dad[30]=-28, và count[28]=37 (chỉ ra nút có trọng 21 sẽ có chỉ mục là 30 và đó là con phải của một cha có chỉ mục 28 và trọng 37). Hình 22.7 cho mảng dad cho các nút nội của cây trong hình 22.5.

Đoạn chương trình sau xây dựng mã Huffman thực sự, như đã được biểu diễn bởi trie trong hình 22.5, từ biểu diễn của cây mã hóa đã được tính trong tiến trình sàng lọc. Mã được biểu diễn bởi hai mảng: code[k] cho biểu diễn nhị phân của ký tự thứ k và len[k] cho số bit của code[k] dùng trong mã. Cho thí dụ, I là chữ thứ 9 và có mã 001, như thế code[9]=1 và len[9]=3.

k	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43
count[k]	60	37	23	21	16	12	11	10	6	6	6	6	5	4	4	3	2
dad[k]	0	-27	27	-28	28	-29	29	30	-31	31	32	-32	-34	-35	36	-36	35

Hình 22.7 *Biểu diễn mối liên kết cha con của cây Huffman*

```
for k:=0 to 26 do
   if count[k]=0 then
      begin code[k]:=0; len[k]:=0 end
   else
      begin
      i:=0; j:=1; t:=dad[k]; x:=0;
      repeat
         if t<0 then begin x:=x+j; t:=-t end;
         t:=dad[t]; j:=j+j; i:=i+1
      until t=0;
      code[k]:=x; len[k]:=i;
      end;
```

Cuối cùng, ta có thể dùng các biểu diễn của mã đã được tính này để mã hoá thông điệp :

```
for j:=1 to M do
   for i:=len[index(a[j])] downto 1 do
      write(bits(code[index(a[j])],i-1,1):1);
```

Chương trình này dùng thủ tục bits từ các chương 10 và 17 để truy xuất các bit đơn lẻ. Thông điệp mẫu của chúng ta được mã hoá chỉ trong 236 bits so với 300 bits được dùng cho việc mã hoá đơn giản, tiết kiệm khoảng 21% :

```
01101111001001101011010110001110011110011101110111001010
11010011101011100111110000011010001001011011001011011100
00011111111011011110100010000011010011010001111000100001
01010010111001011111101000111011101010011101110011
```

Bây giờ, như đã được đề cập ở trên, cây phải được cất hay gửi đi cùng với thông điệp để giải mã nó. May mắn thay, điều này không thể hiện bất kỳ một khó khăn thực sự nào. Thực sự chỉ cần chứa một mảng code, vì trie tìm kiếm theo cơ số (radix search trie), phát sinh từ việc chèn các đầu vào (entry) của mảng đó trong một cây rỗng khởi đầu chính là cây giải mã.

Vì vậy việc tiết kiệm bộ nhớ đã được dẫn chứng ở trên là không hoàn toàn chính xác, vì thông điệp không thể được giải mã mà không có trie và ta phải kết toán phí tổn lưu trie (i.e, mảng code) cùng với thông điệp. Mã hoá Huffman vì vậy chỉ có hiệu quả đối với các tập tin dài ở đó việc tiết kiệm trong thông điệp là đủ để đền

bù lại các phí tổn, hay trong các trường hợp ở đó trie mã hoá có thể được tính toán trước và được dùng cho một số lượng lớn các thông điệp. Ví dụ, một trie dựa trên tần số xuất hiện của các chữ cái trong tiếng Anh có thể được dùng cho các tài liệu văn bản. Đối với vấn đề đó, một trie dựa trên tần số xuất hiện của các ký tự trong các chương trình Pascal có thể được dùng cho các chương trình mã hoá (ví dụ, ";" rất có thể là nằm gần đỉnh của một trie như vậy). Một thuật toán mã hoá Huffman tiết kiệm khoảng 23% khi thực hiện trên văn bản của chương này.

Như trước đây, đối với các tập tin ngẫu nhiên thực sự, thì ngay cả cơ cấu mã hoá khôn ngoan hơn này cũng sẽ không hoạt động được bởi vì mỗi ký tự sẽ xuất hiện xấp xỉ cùng số lần, mà nó sẽ dẫn tới một sự mã hoá cân bằng hoàn toàn và một số lượng bits bằng nhau cho mỗi chữ cái trong mã.

BÀI TẬP

1. Cài đặt các thủ tục nén và bung cho phương pháp mã hoá độ-dài-loạt đối với một bảng mẫu tự cố định được mô tả trong tài liệu, sử dụng Q như một ký tự escape.

2. Liệu "QQ" có thể xảy ra ở đâu đó trong một tập tin đã được nén bởi phương pháp đã được mô tả trong bài hay không ? Có thể xuất hiện "QQQ" hay không ?

3. Cài đặt các thuật toán nén và bung cho phương pháp mã hoá tập tin nhị phân đã được mô tả trong bài.

4. Chữ cái "q" được cho trong bài có thể được xử lý như một dãy các ký tự 5-bit. Hãy thử trình bày các ý kiến thuận và chống về cách làm như vậy để dùng một phương pháp mã hoá độ-dài-loạt dựa-trên-ký-tự (characterd-base).

5. Hãy minh họa tiến trình khởi tạo khi phương pháp đã dùng trong bài học được dùng để xây dựng cây mã hoá Huffman cho chuỗi "ABRACADABRA". Thông điệp đã được mã hoá cần bao nhiêu bits ?

6. Mã Huffman cho một tập tin nhị phân là gì ? Hãy cho một ví dụ minh họa số bits tối đa mà nó có thể được dùng trong một mã Huffman đối với một tập tin tam phân (3-giá trị) N ký tự.

7. Giả sử rằng các tần số xuất hiện của tất cả các ký tự sẽ được mã hoá là khác nhau. Cây mã hoá Huffman có là duy nhất không ?

8. Mã hoá Huffman có thể được mở rộng theo một cách đơn giản để mã hoá thành các ký tự 2-bit (dùng các cây 4-đường (4-way tree)). Ưu điểm và nhược điểm chính của cách làm này là gì ?

9. Kết quả của việc ngắt một chuỗi được-mã-theo-Huffman thành các ký tự 5-bit và giải-mã-Huffman của chuỗi đó là gì ?

10. Cài đặt một thủ tục để giải mã một chuỗi được-mã-theo -Huffman, cho trước các mảng code và len.

23
MẬT MÃ

Trong chương trước ta đã xem xét các phương pháp mã hoá các chuỗi ký tự để tiết kiệm chỗ. Dĩ nhiên, có một lý do khác rất quan trọng để mã hoá các chuỗi ký tự : bảo mật cho chúng.

Mật mã học, nghiên cứu các hệ thống dùng cho việc truyền thông bí mật, gồm có hai lĩnh vực nghiên cứu : Mật mã học (cryptography), thiết kế các hệ truyền thông bí mật, và giải mã học (cryptanalysis), nghiên cứu các phương pháp để giải mã các hệ thống truyền thông bí mật. Mật mã học chủ yếu đã được áp dụng trong các hệ truyền thông quân sự và ngoại giao, nhưng các áp dụng có ý nghĩa khác cũng đang trở nên rõ ràng. Hai ví dụ chính là các hệ thống tập tin máy tính (trong đó mỗi người sử dụng thích giữ riêng các tập tin của họ) và các hệ chuyển ngân điện tử (có liên quan đến những lượng tiền rất lớn). Một người sử dụng máy tính chỉ muốn cất giữ riêng các tập tin máy tính của mình cũng như đã cất các giấy tờ trong tủ hồ sơ, và một ngân hàng thì muốn việc chuyển ngân điện tử sẽ an toàn như là được chuyển ngân bằng xe bọc thép vậy.

Ngoại trừ các ứng dụng quân sự, ta giả định rằng các nhà mật mã học là "loại người tốt" và các nhà giải mã học là "loại người xấu" : mục đích của chúng ta là bảo vệ các tập tin máy tính của mình và các tài khoản ngân hàng chống lại các tội phạm. Nếu quan điểm này có một vẻ gì đó không thân thiện, thì phải lưu ý (mà không quá triết lý) là, bằng cách sử dụng mật mã học, ta đang giả định là có tồn tại những mối quan hệ không thân thiện ! Dĩ nhiên, "loại người tốt" phải biết một cái gì đó về các nhà giải mã học, vì cách tốt nhất để bảo đảm một hệ thống là an toàn là tự mình thử giải mã nó. (Cũng như vậy, có nhiều thí dụ về các cuộc chiến đã qua, trong đó có nhiều sinh mạng đã được cứu sống, là nhờ ở sự thành công của việc giải mã).

Mật mã học có nhiều mối liên hệ gần gũi với khoa học máy tính và với các thuật toán, đặc biệt là các thuật toán số học và xử-lý-chuỗi mà ta đã nghiên cứu. Thật vậy, nghệ thuật (hay khoa học ?) mật mã có một mối liên hệ mật thiết với các máy tính và khoa học máy tính mà nó chỉ mới bắt đầu được hiểu một cách đầy đủ. Giống như các thuật toán, các hệ mật mã đã xuất hiện từ rất lâu đời so với các máy tính. Việc thiết kế hệ thống bảo mật và việc thiết kế thuật toán có một di sản chung, và nó lôi cuốn cùng một người vào cả hai lĩnh vực này.

Không rõ là ngành nào của mật mã học đã bị ảnh hưởng nhiều nhất bởi tính khả dụng của các máy tính. Các nhà giải mã giờ đây có được những cái máy mật mã khả dụng mạnh hơn trước đây nhiều, nhưng giờ đây họ cũng có nhiều chỗ để tạo ra lỗi hơn. Các nhà giải mã học có những công cụ mạnh hơn nhiều để bẻ khoá các mã so với trước đây, nhưng các mã bị bẻ sẽ phức tạp hơn. Các nhà mật mã học có thể phải sử dụng đến một nguồn tài nguyên lớn để tính toán; đó không chỉ là một trong số các lĩnh vực ứng dụng đầu tiên cho các máy tính, mà đó vẫn còn là một lĩnh vực ứng dụng chủ yếu cho các siêu máy tính hiện đại.

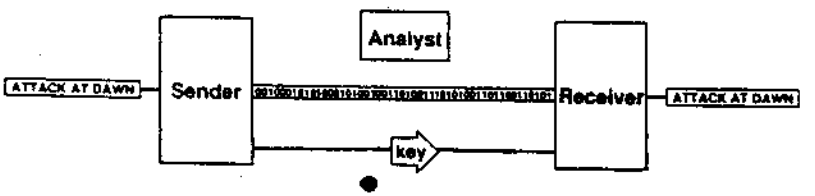
Gần đây hơn, việc sử dụng rộng rãi máy tính đã dẫn tới sự xuất hiện hàng loạt các ứng dụng mới quan trọng cho mật mã học, như đã được đề cập ở trên. Gần đây các phương pháp mật mã mới đã được phát triển phù hợp với các ứng dụng như vậy, và những ứng dụng này đã dẫn tới sự phát hiện ra một mối quan hệ nền tảng giữa mật mã học và một lĩnh vực quan trọng khoa học máy tính lý thuyết mà ta sẽ xem xét một cách ngắn gọn trong chương 45.

Trong chương này, ta sẽ xem xét một vài đặc trưng cơ bản của các thuật toán mật mã. Chúng ta sẽ cố gắng không đào sâu vào việc cài đặt chi tiết: mật mã học là một lĩnh vực đặc thù sẽ được dành cho các chuyên gia. Trong khi không có gì là khó khăn để "giữ cho con người được lương thiện" bằng cách bảo mật mọi thứ với một thuật toán mật mã đơn giản, tuy nhiên thật mạo hiểm khi dựa vào một phương pháp được cài đặt bởi một người-không-phải-chuyên-gia.

QUY LUẬT CỦA TRÒ CHƠI

Các thành phần cơ sở dẫn tới việc cung cấp một phương tiện cho việc truyền thông an toàn giữa hai cá nhân được gọi một cách chọn lọc là một hệ mật mã. Cấu trúc tiêu chuẩn của một hệ mật mã đặc trưng được vẽ trong hình 23.1.

Nơi gửi (sender) gửi đi một thông điệp (được gọi là văn bản thật (plaintext)) cho nơi nhận (receiver) bằng cách biến đổi văn bản thật thành ra một dạng bí mật thích hợp cho việc gửi đi (được gọi là văn bản mã (ciphertext)) bằng cách dùng một thuật toán mật mã (phương pháp mật mã) và một vài tham số chìa khoá (key). Để đọc thông điệp, nơi nhận phải có một thuật toán mật mã tương xứng (phương pháp giải mã) cùng với các tham số chìa khoá, mà nó sẽ biến đổi văn bản mã trở lại thành ra văn bản thật, là thông điệp. Thông thường ta giả định là văn bản mã được gửi trên các đường truyền không an toàn và khả dụng đối với người giải mã. Thường ta cũng giả định là các phương pháp mật mã và giải mã là đều quen thuộc đối với người giải mã : mục đích của anh ta là khôi phục lại văn bản thật từ văn bản mã mà không biết các tham số chìa khoá. Như một quy luật là càng có nhiều tham số chìa khoá thì hệ mật mã lại càng an toàn nhưng lại càng bất tiện khi sử dụng. Trường hợp này cũng giống với các hệ thống mật mã quy ước khác: một sự an toàn tổ hợp thì an toàn hơn với nhiều con số hơn trên khoá tổ hợp, nhưng lại khó nhớ được tổ hợp khóa đó hơn. Song song với các hệ thống quy ước cũng cần nhắc nhở rằng bất kỳ một hệ mật mã nào cũng chỉ là đáng tin cậy ngang với người sở hữu khoá đó.



Hình 23.1 *Một hệ mật mã đặc trưng*

Điều quan trọng là nhớ rằng các vấn đề kinh tế đóng một vai trò trung tâm trong các hệ mật mã. Có một lý do kinh tế để xây dựng các thiết bị mật mã và giải mã đơn giản (vì có thể nhiều thiết bị cần được cung cấp và các thiết bị phức tạp chi phí nhiều hơn). Cũng vậy, có một lý do kinh tế để làm giảm bớt lượng thông tin khoá sẽ được phân phối (vì một phương pháp truyền thông rất an toàn và tốn kém phải được sử dụng). Cân đối giữa chi phí cài đặt các thuật toán và việc phân phối thông tin khoá là lượng tiền người giải mã sẽ phải sẵn lòng bỏ ra để mở khoá hệ thống. Đối với hầu hết các áp dụng, đó là mục tiêu của nhà mật mã học để phát triển một hệ thống chi-phí-thấp với đặc tính là nó sẽ làm cho nhà giải mã học phải tốn kém hơn nhiều so với cái giá mà anh ta sẵn sàng bỏ ra để đọc được thông điệp. Đối với một vài ứng dụng, có thể cần đến một hệ mật mã "an toàn tuyệt đối" : một hệ mà đối với nó ta có thể bảo đảm rằng nhà giải mã học có thể không bao giờ đọc được các thông báo cho dù anh ta có sẵn sàng chi phí như thế nào đi nữa. (Các giải thưởng rất cao trong một vài ứng dụng của mật mã học ám chỉ một cách tự nhiên rằng những lượng tiền rất lớn sẽ được sử dụng cho các nhà giải mã học). Trong thiết kế thuật toán, chúng ta có thể giữ lại thông tin về các chi phí để giúp ta chọn được các thuật toán tốt nhất: trong mật mã học, chi phí đóng một vai trò trung tâm trong tiến trình thiết kế.

CÁC PHƯƠNG PHÁP ĐƠN GIẢN

Trong số các phương pháp đơn giản nhất (và xưa nhất) dùng để bảo mật là mật mã Caesar : nếu một chữ cái trong văn bản thật là ký tự thứ N trong bảng mẫu tự, thì thay nó bằng ký tự thứ (N+K) trong bảng mẫu tự, trong đó K là một số nguyên không đổi nào đó (Caesar đã dùng K=3). Bảng dưới đây cho thấy làm thế nào một thông điệp được mã hoá bằng cách dùng phương pháp này với K=1:

Văn bản thật : ATTACK␣AT␣DAWN
Văn bản mã : BUUBDL␣BUAEB XO

Phương pháp này yếu vì người giải mã chỉ phải đoán giá trị của K: bằng cách thử từng trường hợp một trong 26 cách chọn, anh ta có thể bảo đảm rằng anh ta sẽ đọc được thông điệp.

Một phương pháp tốt hơn nhiều là dùng một bảng tổng quát để định nghĩa sự thay thế sẽ được tạo ra : đối với mỗi chữ cái trong văn bản thật, bảng sẽ cho biết chữ cái nào sẽ được đặt vào trong văn bản mã. Ví dụ, nếu bảng cho sự tương ứng

␣ABC DEF GHI JKLMN OPQRSTUVWXYZ
THE␣ QUI CKB ROWNF XJMPDVRLAZYG

thì thông điệp được mã hoá như sau :

Văn bản thật : ATTACK␣AT␣DAWN
Văn bản mã : HVVH␣OTHVTQHAF

Cách này mạnh hơn nhiều so với mật mã Caesar đơn giản, vì người giải mã sẽ phải thử nhiều bảng hơn (khoảng $27! > 10^{28}$) để bảo đảm đọc được thông điệp. Tuy nhiên, các bộ mã "thay thế đơn giản" giống như bộ mã này thì dễ dàng bị mở do các tần số chữ cái vốn có trong ngôn ngữ. Ví dụ, vì E là ký tự thông thường nhất trong văn bản tiếng anh, người giải mã có thể nhận được một điểm khởi đầu tốt khi đọc thông điệp bằng cách tìm chữ cái thường xuất hiện nhất trong văn bản mã và thay nó bằng E. Trong khi điều này có thể không hẳn là một chọn lựa đúng, nhưng nó lại đặc biệt tốt hơn việc thử tất cả 26 chữ cái một cách mò mẫm. Trường hợp tốt hơn nữa (cho người giải mã) là khi bắt gặp các tổ hợp hai-chữ-cái (digram): các digram cụ thể (như QJ) không bao giờ xảy ra trong văn bản tiếng Anh trong khi những cái khác (như ER) lại rất phổ biến. Bằng cách xét các tần số của các chữ cái và các tổ hợp ký tự, một nhà giải mã học có thể mở khoá một bộ mã thay thế đơn giản một cách rất dễ dàng.

Một cách để khiến cho kiểu tấn công này trở nên khó khăn hơn là dùng nhiều bảng hơn. Một ví dụ đơn giản của điều này là một sự mở rộng của bộ mã Caesar được gọi là bộ mã Vigenere : một khoá lặp nhỏ sẽ được dùng để xác định giá trị của K cho mỗi một ký tự. Ở mỗi bước, chỉ mục của chữ cái khoá được cộng thêm với chỉ mục ký tự văn-bản-thật để xác định chỉ mục ký tự văn-bản-mã. Văn bản mẫu thật của chúng ta, với khoá ABC, được mã như sau:

Khoá : AB CABCABC ABCAB
Văn bản thật : AT TACK␣AT␣DAWN
Văn bản mã : BVWBENACWAFDXP

Cho ví dụ, chữ cái cuối cùng của văn bản mã là P, là chữ cái thứ 16 của bảng mẫu tự, do ký tự văn bản thật tương ứng là N (chữ cái thứ 14), và chữ cái khoá tương ứng là B (chữ cái thứ hai).

Bộ mã Vigenere hiển nhiên có thể được chế tạo một cách phức tạp hơn bằng cách dùng các bảng tổng quát khác cho mỗi một chữ cái của văn bản thật (thay vì chỉ là các độ dời đơn giản). Cũng vậy, hiển nhiên là khoá càng dài, thì bộ mã càng tốt hơn. Thực vậy, nếu khoá dài bằng văn bản thật, ta có bộ mã Vernam, được gọi một cách phổ biến hơn là đệm một-lần (one-time pad). Đây là hệ mật mã an toàn tuyệt đối duy nhất được biết, và nó được dùng một cách công khai cho đường dây nóng Washington-Moscow và những ứng dụng trọng yếu khác. Vì mỗi một ký tự khoá được dùng chỉ một lần, người giải mã có thể không còn cách nào tốt hơn là thử mọi chữ cái chìa khoá có thể có, cho mỗi một vị trí trong thông điệp, một trường hợp vô vọng hiển nhiên vì điều này khó ngang với việc thử tất cả các thông điệp có thể có. Tuy nhiên, dùng mỗi một chữ cái khoá chỉ một lần hiển nhiên dẫn tới một bài toán phân phối khoá dịch vụ, và việc đệm-một-lần sẽ chỉ hữu ích đối với các thông điệp tương đối ngắn, không thường xuyên được gửi đi.

Nếu thông điệp và khoá được mã trong dạng nhị phân, một cơ chế phổ biến hơn để mã hoá từng-vị-trí-một là dùng hàm "hoặc-loại-trừ" (XOR): để mã văn bản thật , "xor" nó (từng bit một) với khoá. Một đặc trưng hấp dẫn của phương pháp này là việc giải mã có cùng thao tác với việc mật mã : văn bản thật là sự XOR của văn bản mã và khoá, nhưng thực hiện lần nữa việc XOR văn bản mã và khóa sẽ trả về văn bản thật. Chú ý rằng việc XOR của văn bản mã và văn bản thật sẽ là khoá. Điều này thoạt đầu có vẻ là đáng ngạc nhiên, nhưng thực sự nhiều hệ mật mã có tính chất là người giải mã có thể phát hiện ra khoá nếu như anh ta biết được văn bản thật.

CÁC MÁY MẬT MÃ / GIẢI MÃ

Nhiều ứng dụng mật mã (ví dụ, các hệ thống tiếng nói dùng cho truyền thông quân sự) có liên quan đến việc truyền đi một lượng lớn dữ liệu, và điều này khiến cho việc đệm-một-lần là không phù hợp. Những cái mà ta cần là xấp xỉ với đệm-một-lần trong đó một

lượng lớn các "chìa-khoá-giả" có thể được tạo ra từ một lượng nhỏ các khoá thật sẽ được phân phối.

Cài đặt thông thường trong những trường hợp như vậy là như sau: một máy mật mã được cung cấp một vài biến mật mã (cryptovariables) (khoá thực) bởi nơi gửi, dùng để tạo ra một dòng bit khoá dài (khoá giả). Việc XOR của các bit này và văn bản thật tạo ra văn bản mã. Nơi nhận, có một cái máy tương tự và các biến mật mã, dùng chúng để tạo ra cùng dòng khoá để XOR dựa trên văn bản mã và để lấy lại được văn bản thật.

Việc tạo khoá trong ngữ cảnh này thì rất giống với việc băm và phát sinh số-ngẫu-nhiên, và các phương pháp được bàn luận trong chương 16 và 35 là thích hợp cho việc phát sinh khoá. Thực vậy, một vài cơ chế được bàn luận trong chương 35 đầu tiên đã được phát triển cho việc sử dụng trong các máy mật mã / giải mã như những cái được mô tả ở đây. Tuy nhiên, bộ phát sinh khoá phải là một cái gì đó phức tạp hơn các bộ phát sinh số ngẫu nhiên, vì có những cách để tấn công các máy đơn giản. Vấn đề là người giải mã có thể dễ dàng nhận được một văn bản thật nào đó (ví dụ, nghe lén một hệ tiếng nói), và do đó nhận được một khoá nào đó. Nếu người giải mã biết đủ về máy, thì khoá có thể cung cấp đủ các chi tiết rắc rối để cho phép các giá trị của tất cả các biến mật mã có thể được mô phỏng và tất cả khoá được tính toán bắt đầu từ điểm đó.

Các nhà mật mã học có nhiều cách để tránh được những vấn đề như vậy. Có một cách là khiến cho một bộ phận kiến trúc của chính máy đó trở thành một biến mật mã. Thông thường ta giả định là người giải mã biết mọi thứ về kiến trúc của máy (có thể là một cái máy bị đánh cắp) ngoại trừ các biến mật mã, nhưng nếu một vài biến mật mã được dùng để "tạo cấu hình" cho máy, thì có thể khó tìm được các giá trị của chúng. Một phương pháp khác được sử dụng phổ biến để làm rối trí người giải mã là bộ mã sinh (product cipher), trong đó hai máy khác nhau được tổ hợp lại để sinh ra một dòng khoá phức tạp (hay để điều khiển lẫn nhau). Phương pháp khác là thay thế không tuyến tính; ở đây việc dịch giữa văn bản thật và văn bản mã được thực hiện trong những đoạn lớn, không phải là từng bit một. Vấn đề thường thấy đối với các phương pháp phức tạp như vậy là chúng có thể là quá phức tạp

ngay cả đối với nhà mật mã học để hiểu được nó, và luôn luôn có khả năng là sự việc có thể trở nên tồi tệ hơn đối với một vài sự lựa chọn các biến mật mã nào đó.

CÁC HỆ MẬT MÃ KHÓA-CÔNG-KHAI (Public-key cryptosystems)

Trong các ứng dụng thương mại như chuyển ngân điện tử và thư tín máy tính (thực sự), bài toán phân phối khoá là nặng nề hơn so với các áp dụng mật mã cổ điển. Khuynh hướng cung cấp các khoá dài mà nó phải được thay đổi thường xuyên cho mọi công dân, trong khi vẫn tiếp tục duy trì cả tính an toàn lẫn hiệu-quả-chi-phí, sẽ cản trở rất nhiều đến việc phát triển các hệ thống như vậy. Tuy nhiên, các phương pháp gần đây được phát triển, sẽ hứa hẹn việc loại bỏ bài toán phân phối khoá một cách hoàn toàn. Những hệ thống như vậy, được gọi là các hệ mật mã khoá công khai, rất có thể sẽ được sử dụng rộng rãi trong tương lai gần. Một trong những điểm nổi bật nhất của các hệ thống này là dựa trên một vài thuật toán số học ta đang nhiên cứu, vì vậy ta sẽ xem xét kỹ hơn nó hoạt động như thế nào.

Ý tưởng trong các hệ mật mã khoá công khai là dùng một "danh bạ điện thoại" các khoá mật mã. Khoá mật mã của mỗi người (ký hiệu bởi P) là tri thức công cộng : khoá của một người có thể được liệt kê, ví dụ như, nằm kế số điện thoại của anh ta trong cuốn danh bạ điện thoại. Mỗi người cũng có một khoá bí mật được dùng để giải mã; khoá bí mật này (được ký hiệu bởi S) không được biết bởi bất kỳ ai khác. Để chuyển đi một thông báo M, người gửi tra ra khoá công khai của người nhận, dùng khoá đó để mã hoá thông điệp, và sau đó gửi thông điệp đi. Ta sẽ ký hiệu thông điệp đã được mã (văn bản mã) là C=P(M). Người nhận dùng khoá giải mã riêng để giải mã và đọc thông điệp. Để cho hệ thống này hoạt động, ít nhất các điều kiện sau đây phải được thỏa mãn :

(i) S(P(M)) = M đối với mọi thông điệp M.
(ii) Tất cả các cặp (S,P) là phân biệt.
(iii) Việc phát sinh S từ P là khó ngang với việc đọc M.
(iv) Cả S lẫn P đều được tính dễ dàng.

Điều kiện đầu tiên là một tính chất mật mã căn bản, hai điều kiện sau cung cấp tính an toàn, và cái thứ tư làm cho hệ thống trở nên dễ dùng.

Lược đồ tổng quát này đã được tóm lược bởi W.Diffie và M.Hellman vào năm 1976, nhưng họ đã không có phương pháp nào thỏa mãn được tất cả các điều kiện này. Một phương pháp như vậy đã được khám phá chẳng bao lâu ngay sau đó bởi R.Rivest, A.Shamir, và L.Adleman. Lược đồ của họ, được biết như là hệ mật mã khoá công khai RSA, dựa trên các thuật toán số học được thực hiện trên các số nguyên rất lớn. Khoá mật mã P là cặp số nguyên (N,p) và khoá giải mã S là cặp số nguyên (N,s), trong đó s là bí mật. Các số này dự kiến sẽ là rất lớn (đặc biệt, N có thể tới 200 chữ số và p và s có thể là 100 chữ số). Các phương pháp mật mã và giải mã thì đơn giản : đầu tiên thông điệp được ngắt thành các số nhỏ hơn N (ví dụ, bằng cách lấy lgN bits ở từng thời điểm từ chuỗi nhị phân tương ứng với mã hoá ký tự của thông điệp). Sau đó các số này được nâng lên lũy thừa một cách độc lập rồi modulo cho N : để mật mã một (mẩu của một) thông điệp M, ta tính $C=P(M)=M^P \bmod N$, và để giải mã một văn bản mã C, ta tính $M=S(C)=C^s \bmod N$. Trong chương 36 ta sẽ nghiên cứu làm thế nào để thực hiện phép tính này khi mà việc tính toán với các số 200-chữ số có thể là gặp trở ngại, tuy nhiên việc ta chỉ cần lấy phần dư sau khi chia cho N hiểu là ta có thể giữ cho các số không trở nên lớn, bất chấp sự kiện là chính M^P và C^S là các số lớn không tưởng tượng nổi.

TÍNH CHẤT 23.1 *Trong các hệ mật mã RSA, một thông điệp có thể được mã trong khoảng thời gian tuyến tính.*

Đối với các thông điệp dài, độ dài của các số được dùng cho các khoá có thể được coi như là hằng-một chi tiết cho cài đặt. Tương tự, nâng một số lên một lũy thừa thì được thực hiện trong khoảng thời gian hằng, vì các số không được phép dài hơn một độ dài "hằng". Thực ra, tham số này che giấu nhiều chi tiết cài đặt có liên quan đến việc tính toán với các con số dài : chi phí của các phép toán này thực sự là một yếu tố ngăn cản sự phổ biến khả năng ứng dụng của phương pháp.

Điều kiện (iv) ở trên vì vậy được thỏa mãn, và điều kiện (ii) có thể dễ dàng bị ép thỏa. Ta vẫn phải tiếp tục bảo đảm rằng các biến mật mã N,p và s có thể được chọn sao cho có thể thoả (i) và (iii). Để các điều kiện này trở nên có sức thuyết phục hơn thì phải cần đến sự trình bày về lý thuyết số mà nó nằm ngoài phạm vi của quyển sách này, nhưng chúng ta có thể tóm lược các ý chính. Đầu tiên, cần tạo ra 3 số nguyên tố "ngẫu nhiên" lớn (xấp xỉ 100-chữ số) : số lớn nhất sẽ là s và ta sẽ gọi hai số còn lại là x và y. Sau đó N được chọn sẽ là tích của x và y, và p thì được chọn sao cho ps mod (x-1)(y-1) = 1. Có thể chứng minh rằng, với N,p và s được chọn theo cách này, ta có M^{ps} mod N = M đối với tất cả các thông điệp M.

Ví dụ, với việc mã hoá chuẩn của chúng ta, thông điệp ATTACK AT DAWN có thể tương ứng với một số nguyên 28-chữ số

01202001031100012000004012314

vì A là chữ cái đầu tiên (01) trong bảng mẫu tự, T là chữ cái thứ 20 (20), ... Bây giờ, để giữ cho ví dụ là nhỏ, ta bắt đầu với một vài số nguyên tố 2-chữ số (thay vì 100-chữ số, như đã yêu cầu) : lấy x=47, y=79, và s=97. Những giá trị này dẫn tới N=3713 (tích của x và y) và p=37 (số nguyên duy nhất mà nó cho phần dư là 1 khi được nhân cho 37 và chia cho 3588). Bây giờ, để mã hoá thông điệp, ta ngắt nó thành ra từng đoạn 4-số và nâng lên lũy thừa p (modulo N). Điều này cho ra bản đã được mã hoá

14042932353600013284228022 35

Nghĩa là, $0120^{37} \equiv 1404$, $2001^{37} \equiv 2932$, $0311^{37} \equiv 3536$ (mod 3713), ... Tiến trình giải mã cũng giống như vậy, dùng s thay cho p. Vì vậy, ta nhận được trở lại thông điệp gốc vì $1404^{97} \equiv 0120$, $2932^{97} \equiv 2001$ (mod 3713), ...

Phần quan trọng nhất của các con tính có liên quan đến việc mã hoá thông điệp, như đã bàn trong Tính chất 23.1 ở trên. Nhưng sẽ không có hệ mật mã nào hết nếu như không thể tính được các biến chìa khoá. Cho dù điều này có ám chỉ cả lý thuyết số tinh vi lẫn các chương trình tinh xảo để thao tác trên các số lớn,

thời gian để tính toán các khoá có lẽ là nhỏ hơn bình phương độ dài của chúng (và không tỉ lệ với độ lớn của chúng, mà điều đó là không thể chấp nhận được).

TÍNH CHẤT 23.2 *Các khóa cho hệ mật mã RSA có thể được tạo ra mà không phải tính toán quá nhiều.*

Một lần nữa, ta lại cần đến một vài phương pháp dựa trên lý thuyết số mà nó nằm ngoài phạm vi của cuốn sách này; mỗi số nguyên tố lớn có thể được phát sinh bằng cách đầu tiên tạo ra một số ngẫu nhiên lớn, sau đó kiểm tra các số kế tiếp bắt đầu từ điểm đó cho tới khi tìm được một số nguyên tố. Một phương pháp đơn giản thực hiện một phép tính trên một con số ngẫu nhiên, mà, với xác suất là 1/2, sẽ "chứng minh" rằng số được kiểm tra là không nguyên tố. Bước cuối cùng là tính p : có lẽ chỉ cần đến một biến thể của thuật toán Euclid (xem chương 1).

Nhớ lại là khoá giải mã s (và các thừa số x và y của N) sẽ được giữ bí mật, và sự thành công của phương pháp tùy thuộc vào việc người giải mã không có khả năng để tìm giá trị của s, nếu cho trước N và p. Bây giờ, đối với ví dụ nhỏ của chúng ta, dễ phát hiện ra rằng 3713 = 47*79, nhưng nếu N là một số 200-chữ số, thì có ít hy vọng tìm ra các thừa số của nó. Nghĩa là, có vẻ khó tìm ra được s từ tri thức về p (và N), mặc dù không ai có thể chứng minh điều đó. Rõ ràng, tìm p từ s cần tri thức về x và y, và rõ ràng cần phân tích N ra thừa số để tính x và y. Nhưng việc phân tích N ra thừa số là rất khó : các thuật toán phân tích ra thừa số tốt nhất đã được biết cần hàng triệu năm để phân tích một số 200-chữ số ra thừa số, dùng kỹ thuật hiện có.

Một đặc trưng hấp dẫn của hệ RSA là các tính toán phức tạp ám chỉ N,p và s được thực hiện chỉ một lần cho mỗi một người sử dụng có đăng ký với hệ thống, trong khi đó thì rất nhiều thao tác mật mã và giải mã thông thường chỉ ngắt thông điệp và áp dụng thủ tục lũy thừa đơn giản. Sự đơn giản về mặt tính toán này, kết hợp với tất cả các đặc trưng thuận lợi được cung cấp bởi các hệ mật mã khoá-công-khai, đã khiến cho hệ thống này thật hấp dẫn cho việc truyền thông an toàn, đặc biệt trên các hệ thống máy tính và mạng máy tính.

Phương pháp RSA có nhược điểm của nó : thủ tục lũy thừa thực sự tốn kém bởi các tiêu chuẩn mật mã, và tệ hơn là, về lâu về dài, có khả năng đọc được các thông điệp đã được mật mã bằng cách dùng phương pháp này. Điều này là đúng đối với nhiều hệ mật mã : một phương pháp mật mã phải chống lại được các cuộc tấn công để giải mã một cách nghiêm túc trước khi nó có thể được dùng với sự tin tưởng.

Nhiều phương pháp khác đã được đề nghị để cài đặt các hệ mật mã khoá-công-khai. Một vài phương pháp thú vị nhất đã được gắn với một lớp bài toán quan trọng mà nó thường được xem là rất khó (mặc dù điều này thì không được biết chắc lắm), mà ta sẽ nghiên cứu trong chương 45. Các hệ mật mã này có một tính chất thú vị là một cuộc tấn công thành công có thể cung cấp một cái nhìn sâu hơn về cách làm thế nào để giải được một vài bài toán khó nổi tiếng chưa giải được (như việc phân tích ra thừa số đối với phương pháp RSA). Việc liên kết giữa mật mã học với các chủ đề cơ sở trong nghiên cứu khoa học máy tính, cùng với khả năng sử dụng rộng rãi mật mã khoá-công-khai, đã khiến cho lãnh vực này trở thành một môi trường nghiên cứu rất năng động hiện nay.

BÀI TẬP

1. Giải mã thông điệp sau đây, đã được mã bằng một bộ mã Vigenere sử dụng mẫu CAB (lặp lại nếu cần) cho khoá (trên một bảng mẫu tự 27-chữ-cái, với ký tự khoảng trống nằm trước A) : DOBHBUAASXFZWJQQ

2. Bảng nào sẽ được dùng để giải mã các thông điệp mà nó đã được mã bằng cách dùng phương pháp thay thế theo bảng ?

3. Giả sử rằng một bộ mã Vigenere với một khoá 2-ký-tự được sử dụng để mã hoá một thông điệp tương đối dài. Viết một chương trình để suy ra khoá, dựa trên giả định là tần số xuất hiện của mỗi ký tự trong các vị trí lẻ sẽ là gần bằng với tần số xuất hiện của mỗi ký tự trong các vị trí chẵn.

4. Viết các thủ tục mật mã và giải mã tương xứng mà nó dùng đến phép toán "xor" giữa một bản nhị phân của thông điệp và một dòng nhị phân từ một trong số các bộ phát sinh số ngẫu nhiên điều hoà tuyến tính của chương 3.

5. Viết một chương trình để chẻ nhỏ phương pháp đã cho trong bài tập trước, giả sử rằng 10 ký tự đầu tiên của thông điệp đã biết là ký tự khoảng trống.

6. Người ta có thể mã hóa một văn bản thật bằng cách "and" ("và") nó (từng bit một) với khoá được không ? Giải thích tại sao được hay tại sao không ?

7. Đúng hay sai ? Mật mã khoá-công-khai thuận tiện để gửi cùng thông điệp tới nhiều người sử dụng khác nhau. Bàn luận về câu trả lời của bạn.

8. P(S(M)) là gì đối với phương pháp RSA cho mật mã khoá - công -khai ?

9. Mật mã RSA có thể ám chỉ việc tính M^n, trong đó M có thể là một số k-chữ-số được biểu diễn trong một mảng k số nguyên. Cần khoảng bao nhiêu phép toán cho phép tính này ?

10. Cài đặt các thủ tục mật mã /giải mã cho phương pháp RSA (giả định rằng s,p và N tất cả đều được cho trước và được biểu diễn trong các mảng số nguyên có kích thước 25).

MỤC LỤC

Tập 1- CÁC THUẬT TOÁN THÔNG DỤNG

Tập 2 - CÁC THUẬT TOÁN CHUYÊN DỤNG

PHẦN 5 - CÁC THUẬT TOÁN HÌNH HỌC

PHẦN 6 - CÁC THUẬT TOÁN ĐỒ THỊ

PHẦN 7 - CÁC THUẬT TOÁN TOÁN HỌC

PHẦN 8 - CÁC ĐẶC TRƯNG CẤP CAO

ROBERT SEDGEWICK

CẨM NANG
THUẬT TOÁN

Tập 1 : CÁC THUẬT TOÁN THÔNG DỤNG

Chịu trách nhiệm xuất bản Pts. TÔ ĐĂNG HẢI
Người dịch TRẦN ĐAN THƯ
VŨ MẠNH TƯỜNG
DƯƠNG VŨ DIỆU TRÀ
NGUYỄN TIẾN HUY
Hiệu đính Gs.Ts. HOÀNG KIẾM
Biên tập TUẤN NGHĨA
Trình bày bìa THU THỦY

6T7.3
KHKT-2006 72-2005/CXB/75-39/KHKT

NHÀ XUẤT BẢN KHOA HỌC VÀ KỸ THUẬT
70 Trần Hưng Đạo - Hà Nội
Chi nhánh : 28 Đồng Khởi Q.1 TP.HCM

In 1000 cuốn, khổ 14,5 x 20,5 cm tại Xưởng in Ban TT - VH Thành ủy TP.HCM. Giấy QĐXB số: 72-2005/CXB/75-39/KHKT cấp ngày 07/11/2005 NXB KHKT.
In xong và nộp lưu chiểu tháng 1/2006.

Niklaus Wirth said that

Data Structures

\+ Algorithms

\= Programs

8 935048 953648

Giá: 36.000đ

ALGORITHMS ALGORITHMS ALGORITHMS ALGORITHMS