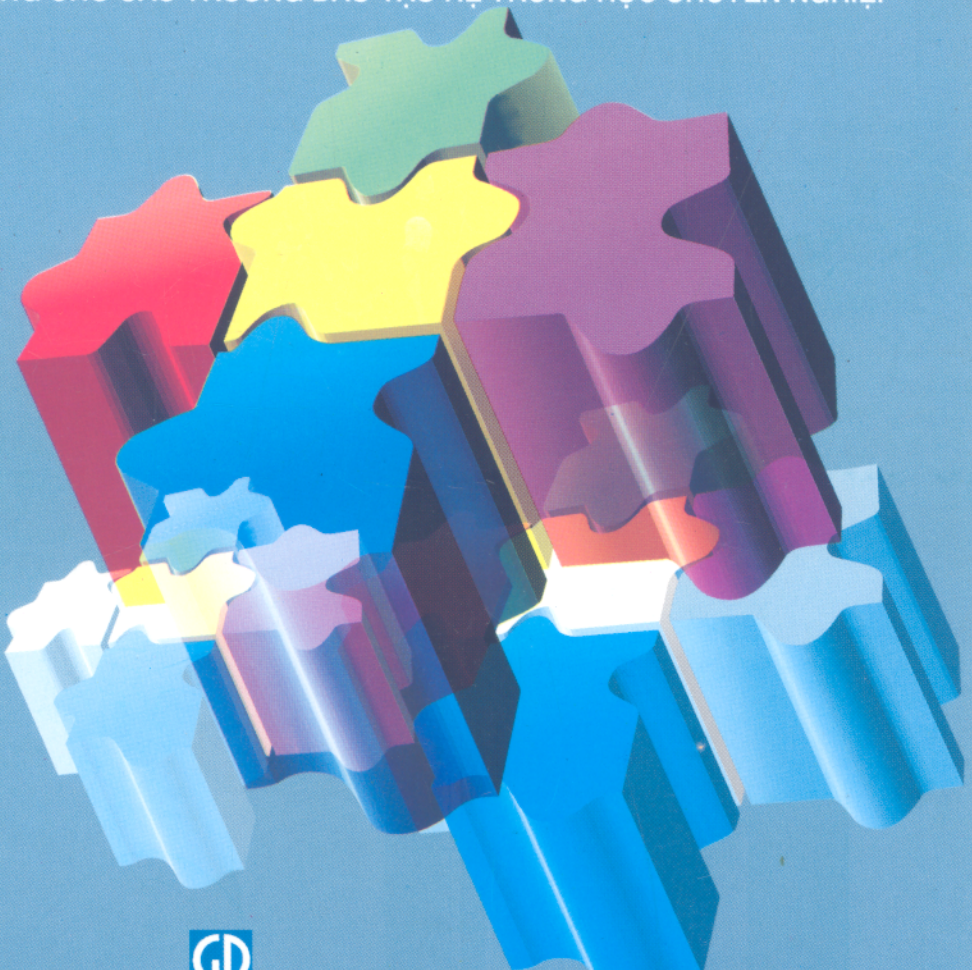
VỤ GIÁO DỤC CHUYÊN NGHIỆP

GIÁO TRÌNH
CẤU TRÚC DỮ LIỆU VÀ GIẢI THUẬT

SÁCH DÙNG CHO CÁC TRƯỜNG ĐÀO TẠO HỆ TRUNG HỌC CHUYÊN NGHIỆP



NHÀ XUẤT BẢN GIÁO DỤC

PGS. TS. ĐỖ XUÂN LÔI

Giáo trình
CẤU TRÚC DỮ LIỆU và GIẢI THUẬT

(Sách dùng cho các trường Đào tạo hệ Trung học chuyên nghiệp)

(Tái bản lần thứ hai)

NHÀ XUẤT BẢN GIÁO DỤC

135.833
Xếp

04 - 2006/CXB/33-1860/GD Mã số: 6H160T6 - DAI

Lời giới thiệu

Năm 2002, Vụ Giáo dục Chuyên nghiệp – Bộ Giáo dục và Đào tạo đã phối hợp với Nhà xuất bản Giáo dục xuất bản 21 giáo trình phục vụ cho đào tạo hệ THCN. Các giáo trình trên đã được nhiều trường sử dụng và hoan nghênh. Để tiếp tục bổ sung nguồn giáo trình đang còn thiếu, Vụ Giáo dục Chuyên nghiệp phối hợp cùng Nhà xuất bản Giáo dục tiếp tục biên soạn một số giáo trình, sách tham khảo phục vụ cho đào tạo ở các ngành : Điện – Điện tử, Tin học, Khai thác cơ khí. Những giáo trình này trước khi biên soạn, Vụ Giáo dục Chuyên nghiệp đã gửi đề cương về trên 20 trường và tổ chức hội thảo, lấy ý kiến đóng góp về nội dung đề cương các giáo trình nói trên. Trên cơ sở nghiên cứu ý kiến đóng góp của các trường, nhóm tác giả đã điều chỉnh nội dung các giáo trình cho phù hợp với yêu cầu thực tiễn hơn.

Với kinh nghiệm giảng dạy, kiến thức tích luỹ qua nhiều năm, các tác giả đã cố gắng để những nội dung được trình bày là những kiến thức cơ bản nhất nhưng vẫn cập nhật được với những tiến bộ của khoa học kỹ thuật, với thực tế sản xuất. Nội dung của giáo trình còn tạo sự liên thông từ Dạy nghề lên THCN.

Các giáo trình được biên soạn theo hướng mở, kiến thức rộng và cố gắng chỉ ra tính ứng dụng của nội dung được trình bày. Trên cơ sở đó tạo điều kiện để các trường sử dụng một cách phù hợp với điều kiện cơ sở vật chất phục vụ thực hành, thực tập và đặc điểm của các ngành, chuyên ngành đào tạo.

Để việc đổi mới phương pháp dạy và học theo chỉ đạo của Bộ Giáo dục và Đào tạo nhằm nâng cao chất lượng dạy và học, các trường cần trang bị đủ sách cho thư viện và tạo điều kiện để giáo viên và học sinh có đủ sách theo ngành đào tạo. Những giáo trình này cũng là tài liệu tham khảo tốt cho học sinh đã tốt nghiệp cần đào tạo lại, nhân viên kỹ thuật đang trực tiếp sản xuất.

Các giáo trình đã xuất bản không thể tránh khỏi những sai sót. Rất mong các thầy, cô giáo, bạn đọc góp ý để lần xuất bản sau được tốt hơn. Mọi góp ý xin gửi về : Công ty Cổ phần sách Đại học – Dạy nghề, 25 Hàn Thuyên – Hà Nội.

VỤ GIÁO DỤC CHUYÊN NGHIỆP - NXB GIÁO DỤC

Lời nói đầu

Tìm hiểu về "Cấu trúc dữ liệu và giải thuật" là một đòi hỏi cần thiết đối với những người làm tin học khi muốn tiếp cận với việc lập trình để giải bài toán trên máy tính điện tử, cũng như khi muốn đi sâu thêm vào các lĩnh vực kiến thức khác của công nghệ thông tin.

Trong khuôn khổ một giáo trình nhập môn ở đây tác giả chỉ giới thiệu một số kiến thức cơ sở, bao hàm trong 6 chương.

Chương 1 trình bày khái quát một số khái niệm có liên quan tới giải thuật.

Từ chương 2 đến chương 6, giới thiệu những cấu trúc dữ liệu phổ dụng. Mỗi cấu trúc đã được minh hoạ cụ thể, được nêu rõ cách cài đặt trong máy tính và được thể hiện vai trò qua các bài toán áp dụng thực tế.

Rất có thể, có những khái niệm chỉ được trình bày theo một nghĩa hẹp nào đó, đấy là một dụng ý của tác giả nhằm làm cho việc giới thiệu các nội dung được đơn giản hơn, dễ tiếp nhận hơn, tránh sa đà vào những chi tiết chưa cần thiết, ngay ở những bước "mở đầu".

Phần câu hỏi và bài tập cũng được lựa chọn ở mức trung bình, vừa đủ để kiểm tra việc tiếp nhận những kiến thức đã được giới thiệu, vừa góp phần bổ sung thêm một số khái niệm mới và kĩ năng giải quyết bài toán.

Phần hướng dẫn giải bài tập và lời giải một số bài tập chọn lọc, được nêu ở cuối sách, có mục đích gợi ý thêm về phương hướng hoặc cung cấp một đáp án, giúp cho người đọc có thể tham khảo, đối chiếu khi tự mình làm bài.

Cuốn sách có thể được dùng làm tài liệu học tập cho học viên các lớp trung cấp tin học, cao đẳng tin học hoặc làm tài liệu tham khảo cho sinh viên các ngành khi bước đầu muốn nâng cao trình độ về tin học của mình.

Hà Nội ngày 15 tháng 5 năm 2004

Tác giả

Chương 1. GIẢI THUẬT

1.1. CẤU TRÚC DỮ LIỆU VÀ GIẢI THUẬT

Giải thuật là một khái niệm cơ sở của tin học.

Thuật ngữ "algorithm", nghĩa là "giải thuật" (hay thuật toán) xuất phát từ tên một nhà toán học Ả Rập : Abu Já far Mohammed ibn Musa al *Khowarizmi* (năm 825 sau công nguyên) người đã viết một cuốn sách trong đó có mô tả về cách tính toán.

Giải thuật thể hiện một giải pháp cụ thể, thực hiện từng bước một, để đưa tới lời giải cho một bài toán nào đó.

Có thể nói : giải thuật là một tập hữu hạn các phép toán cơ sở, được sắp đặt theo những quy tắc chính xác, nhằm giải một bài toán.

Các phép toán cơ sở là những phép toán *đơn giản* mà thời gian thực hiện nó luôn là một hằng số, nghĩa là nó không phụ thuộc gì vào kích thước của toán hạng.

Các phép toán trong giải thuật luôn được *xác định rõ ràng*, không mập mờ, ai cũng có thể hiểu được cách thực hiện nó và chỉ một cách duy nhất.

Với một bộ dữ liệu của bài toán, giải thuật sẽ kết thúc *sau một số hữu hạn bước* và cho một lời giải.

Khi giải một bài toán trên máy tính điện tử (MTĐT) ta quan tâm ngay đến việc thiết kế giải thuật. Nhưng cần nhớ rằng : giải thuật là đặc trưng cho *cách xử lí*, mà cách xử lí thì thường liên quan tới *đối tượng xử lí*, tức là "dữ liệu". Cung cách thể hiện dữ liệu mà theo đó chúng được lưu trữ và được xử lí trong MTĐT, được gọi là *cấu trúc dữ liệu*.

Hình dung và tổ chức các dữ liệu theo cấu trúc nào điều đó có ảnh hưởng tới cách xử lí. Như vậy giữa cấu trúc dữ liệu và giải thuật luôn có quan hệ ; thay đổi cấu trúc dữ liệu sẽ dẫn đến thay đổi giải thuật.

Chẳng hạn : Xét một danh mục điện thoại có dạng $<a_i, b_i>$ mà $1 \le i \le n$, với a_i là kí hiệu chỉ họ tên người "thuê bao", b_i chỉ "số điện thoại" . Chúng ta muốn tìm số điện thoại của một người có họ tên X nào đó.

Nếu danh mục điện thoại được ghi chép tự nhiên trong sổ tay của ta thì việc đi tìm người thuê bao có họ tên X để truy ra số điện thoại của họ, chỉ có thể thực hiện bằng cách so sánh X với a_i với i = 1, 2, 3...v.v. cho tới khi hoặc gặp một a_k = X thì truy ra được số điện thoại b_k tương ứng ; hoặc không tìm ra được, sau khi đã duyệt hết cả danh sách.

Như vậy là ta đã thực hiện một giải thuật tìm kiếm được gọi là "*tìm kiếm tuần tự*" (sequential search).

Nhưng nếu danh mục điện thoại lại được tổ chức sắp xếp theo thứ tự từ điển (giống như sắp xếp các từ trong từ điển) thì việc đi tìm số điện thoại của X giống như việc đi tìm nghĩa của một từ mà ta cần tra cứu.

Trong trường hợp này không bao giờ ta lại áp dụng giải thuật "tìm kiếm tuần tự" như đã nêu ở trên cả !

Rõ ràng "giải thuật " đã thay đổi, khi "cấu trúc dữ liệu " thay đổi.

Mỗi ngôn ngữ lập trình đều ấn định sẵn những cấu trúc dữ liệu riêng cho mình : đó là các cấu trúc dữ liệu *tiền định*, chúng được thể hiện qua các *kiểu dữ liệu* của ngôn ngữ đó. Thường đa số các cấu trúc tiền định này là các cấu trúc dữ liệu thông dụng. Ngoài ra có thể có những cấu trúc dữ liệu đặc biệt có ở ngôn ngữ này mà không có ở ngôn ngữ khác. "Người dùng" (user), khi sử dụng một ngôn ngữ nào để thể hiện một giải thuật giải bài toán của mình, phải biết linh hoạt tổ chức dữ liệu của bài toán theo các cấu trúc tiền định của ngôn ngữ đó. Rất có thể, ngôn ngữ đang sử dụng không có sẵn các cấu trúc thật khớp với dữ liệu của bài toán, việc vận dụng khéo léo các cấu trúc hiện có của ngôn ngữ để biểu diễn cấu trúc riêng cho dữ liệu thuộc bài toán của mình, hoàn toàn phụ thuộc vào khả năng và kĩ xảo của "người dùng" !

1.2. NGÔN NGỮ DIỄN ĐẠT GIẢI THUẬT

Mặc dầu vấn đề ngôn ngữ lập trình không được đặt ra ở cuốn sách này, nhưng để diễn đạt các giải thuật mà ta sẽ trình bày dưới đây, ta cũng phải lựa chọn một ngôn ngữ. Có thể nghĩ ngay đến việc sử dụng một ngôn ngữ cấp cao hiện có, chẳng hạn như : PASCAL, C, v. v..., nhưng như vậy sẽ gặp một số hạn chế :

– Phải luôn luôn tuân thủ các quy tắc chặt chẽ về cú pháp của ngôn ngữ đó, khiến cho việc trình bày về giải thuật và cấu trúc dữ liệu có thiên hướng nặng nề, gò bó.

– Phải phụ thuộc vào cấu trúc dữ liệu tiền định của ngôn ngữ, nên có lúc không thể hiện được đầy đủ các ý về cấu trúc mà ta mong muốn giới thiệu.

– Ngôn ngữ đã chọn không phải ai cũng ưa thích và sử dụng.

Vì vậy ở đây ta sẽ dùng một ngôn ngữ "thô hơn" có đủ khả năng diễn đạt được giải thuật trên các cấu trúc để cập đến (mà ta giới thiệu bằng Tiếng Việt), với một mức độ linh hoạt nhất định, không quá gò bó, không câu nệ nhiều về cú pháp nhưng cũng gần gũi với các ngôn ngữ chuẩn để việc chuyển đổi khi cần thiết được dễ dàng. Ta tạm gọi bằng cái tên : "ngôn ngữ tựa PASCAL". Sau đây là một số quy tắc bước đầu. Ở các phần sau sẽ có thể bổ sung thêm.

1.2.1. Quy cách về cấu trúc chương trình

Mỗi chương trình đều được gán một tên để phân biệt, tên này được viết bằng chữ in hoa, có thể có thêm dấu gạch nối và bắt đầu bằng từ khoá **Program**

Ví dụ : **Prorgram** NHAN–MA–TRAN

Độ dài tên không hạn chế.

Sau tên có thể kèm theo lời thuyết minh (ở đây ta quy ước dùng Tiếng Việt) để giới thiệu tóm tắt nhiệm vụ của giải thuật hoặc một số chi tiết cần thiết. Phần thuyết minh được đặt giữa hai dấu {...}.

Chương trình bao gồm nhiều bước, mỗi bước được phân biệt bởi số thứ tự, có thể kèm theo những lời thuyết minh.

1.2.2. Kí tự và biểu thức

a) Kí tự dùng ở đây cũng giống như trong các ngôn ngữ chuẩn, nghĩa là gồm :

– 26 chữ cái Latinh in hoa hoặc in thường

– 10 chữ số thập phân

– Các dấu phép toán số học +, – , *, /, ↑ (lũy thừa)

– Các dấu phép toán quan hệ

$$<, =, >, \leq, \geq, \neq.$$

– Giá trị logic : **true, false**

– Dấu phép toán logic :

and, or, not

– Tên biến : dãy chữ cái và chữ số, bắt đầu bằng chữ cái

– Biến chỉ số có dạng :

A[i], B[i,j] v.v....

b) Còn biểu thức cũng như thứ tự ưu tiên của các phép toán trong biểu thức cũng theo quy tắc như trong PASCAL hay các ngôn ngữ chuẩn khác.

1.2.3. Các câu lệnh (hay các chỉ thị)

Các câu lệnh trong chương trình được viết cách nhau bởi dấu chấm phẩy chúng bao gồm :

1. Câu lệnh gán

Có dạng V : = E

Với V chỉ tên biến, tên hàm ;

E chỉ biểu thức.

Ở đây cho phép dùng phép gán chung.

Ví dụ : A : = B : = 0.1.

2. Câu lệnh ghép

Có dạng :

begin S_1 ; S_2 ; ... ; S_n **end**

Với S_i i = 1, ..., n là các câu lệnh

Nó cho phép ghép nhiều câu lệnh lại để được coi như một câu lệnh.

3. Câu lệnh điều kiện

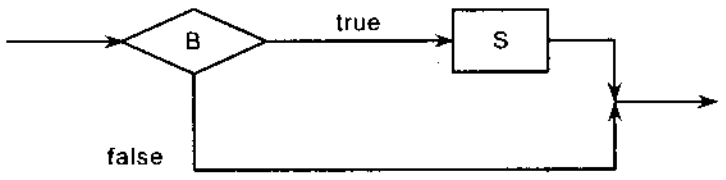
Có dạng :

if B **then** S

Với B là biểu thức logic ;

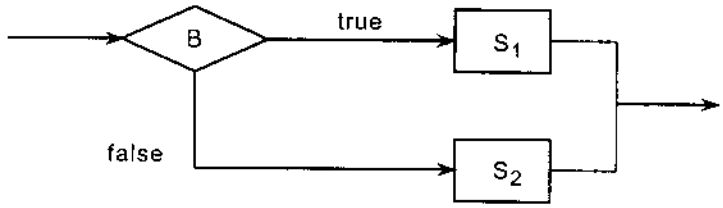
S là một câu lệnh khác.

Có thể diễn tả bởi sơ đồ :



hoặc

if B **then** S_1 **else** S_2



4. Câu lệnh tuyến

Case

B_1 : S_1 ;

B_2 : S_2 ;

...

...

B_n : S_n ;

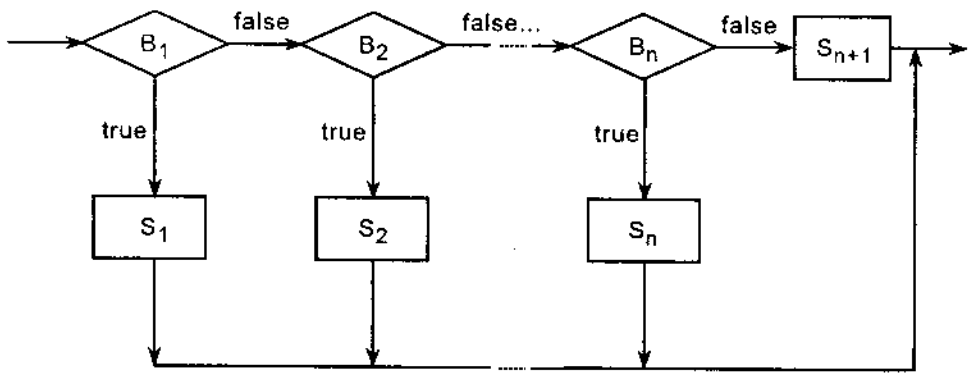
else : S_{n+1}

end case

Với B_i (i = 1, 2, ..., n) là các điều kiện ;

S_i (i = 1, 2, ..., n) là các câu lệnh.

a) Câu lệnh này cho phép phân biệt các tình huống xử lí khác nhau trong các điều kiện khác nhau mà không phải dùng tới các câu lệnh **if – then – else** lồng nhau. Có thể diễn tả bởi sơ đồ :



b) Vài điểm linh động

else có thể không có mặt.

S_i (i = 1, 2,..., n) có thể được thay thế bằng một dãy các câu lệnh thể hiện một dãy xử lí khi có điều kiện B_i mà không cần phải đặt giữa : **begin** và **end**

5. Câu lệnh lặp

a) Với số lần lặp biết trước

for i : = m **to** n **do** S

nhằm thực hiện câu lệnh S với i lấy giá trị nguyên từ m tới n (n ≥ m) với bước nhảy tăng bằng 1,

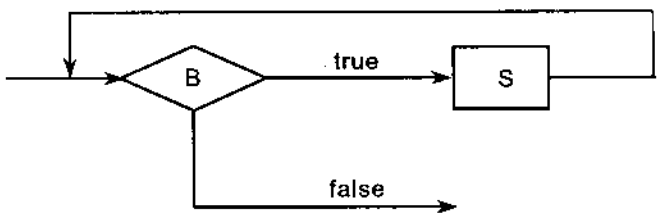
hoặc :

for i : = n **down to** m **do** S

tương tự như câu lệnh trên với bước nhảy giảm bằng 1.

b) Với số lần lặp không biết trước

While B **do** S

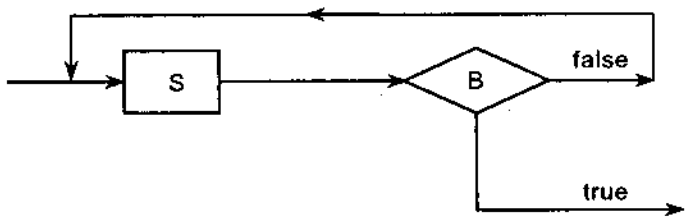


Chừng nào mà B có giá trị bằng true thì thực hiện S

hoặc :

repeat S **until** B

Lặp lại S cho tới khi B có giá trị true (S có thể là một dãy lệnh)



6. Câu lệnh vào, ra

Có dạng :

read (<danh sách biến>)

write (<danh sách biến hoặc dòng kí tự>)

các biến trong danh sách cách nhau bởi dấu phảy.

Dòng kí tự là một dãy các kí tự đặt giữa hai dấu nháy ' '.

7. Câu lệnh kết thúc chương trình

end

1.2.4. Chương trình con

1. Chương trình con hàm

Có dạng :

function <tên hàm> (<danh sách tham số>)

S_1 ; S_2 ; ... ; S_n

return

Câu lệnh kết thúc chương trình ở đây là **return** thay cho **end.**

2. Chương trình con thủ tục

Tương tự như trên, chỉ khác ở chỗ :

Từ khoá **procedure** thay cho **function**.

Trong cấu tạo của chương trình con hàm bao giờ cũng có câu lệnh gán mà tên hàm nằm ở vế trái. Còn đối với chương trình con thủ tục thì không có.

Lời gọi chương trình con hàm thể hiện bằng tên hàm cùng danh sách tham số thực sự, nằm trong biểu thức. Còn với chương trình con thủ tục lời gọi được thể hiện bằng câu lệnh **call** có dạng :

Call <tên thủ tục> (<danh sách tham số thực sự>)

Chú ý : Trong các chương trình diễn đạt một giải thuật ở đây phần khai báo dữ liệu được bỏ qua. Nó được thay bởi phần mô tả cấu trúc dữ liệu bằng ngôn ngữ tự nhiên, mà ta sẽ nêu ra trước khi bước vào giải thuật.

Như vậy nghĩa là các chương trình được nêu ra chỉ là đoạn thể hiện các phép xử lí theo giải thuật đã định, trên các cấu trúc dữ liệu được mô tả trước đó, bằng ngôn ngữ tự nhiên.

1.3. THIẾT KẾ GIẢI THUẬT

Tạo lập giải thuật để giải một bài toán, là một nghệ thuật mà không bao giờ có thể nêu đầy đủ ngay một lúc được.

Có nhiều phương pháp thiết kế giải thuật khác nhau, thông dụng là cách thiết kế kiểu "top – down" : cách thiết kế "đi từ tổng thể đến chi tiết". Chiến thuật được áp dụng để thể hiện cách thiết kế này là chiến thuật "chia để trị" nghĩa là tách bài toán ra thành các bài toán con (thành các mô–đun : mô–đun hóa). Với mỗi bài toán con này lại áp dụng một chiến thuật tương tự, cho tới khi đi tới những bài toán con đủ nhỏ để có thể giải trực tiếp được. Sau đó chỉ cần tổng hợp lại các phép xử lí để có giải thuật của bài toán gốc.

Để làm được những điều đó, đứng trước một bài toán, thông thường ta phải :

– Xác định được rõ dữ liệu và yêu cầu : cho biết cái gì ? (dữ liệu input) và đòi hỏi cái gì ? (dữ liệu output).

– Để giải quyết được yêu cầu thì "*phải làm gì ?*" : ở đây mới chỉ phân hoạch được công việc và xác định được mục tiêu của công việc đó.

– Với mỗi công việc ấy thì "*phải làm thế nào*" ?

Trên cơ sở đó mới cụ thể hoá dần dần các phép xử lí để xây dựng giải thuật cần thiết.

Tất nhiên, khi giải quyết câu hỏi "làm thế nào ?" thì dữ liệu input cũng phải được định hình về cấu trúc.

Ví dụ, ta xét bài toán :

Sắp xếp một dãy số (a_1, a_2, ..., a_n) thành một dãy số tăng dần.

• Như vậy dãy số input, nếu có dạng, chẳng hạn :

(33, 77, 11, 55, 99, 22, 44, 88, 66)

thì dãy số output phải có dạng :

(11, 22, 33, 44, 55, 66, 77, 88, 99)

• Để có được kết quả output như vậy thì *phải làm gì* ?

Có thể thấy rằng : sắp xếp theo thứ tự tăng dần nghĩa là :

– Số bé nhất trong n số phải được đặt vào vị trí đầu tiên.

– Số bé nhất trong (n – 1) số còn lại phải được đặt vào vị trí thứ hai.

v.v....

Như vậy sẽ có hai công việc chính phải làm :

• Chọn số bé nhất trong dãy số chưa được sắp.

• Đặt nó vào vị trí sau phần tử cuối của dãy số đã được sắp (nó lại trở thành phần tử cuối cho bước tiếp theo).

Chú ý rằng : lúc đầu dãy số được sắp còn rỗng, sau đó nó được bổ sung dần dần các phần tử vào.

Các công việc trên sẽ được lặp lại (n – 1) lần : lần đầu với n số, lần cuối với 2 số.

• Để thực hiện được hai công việc nêu trên thì phải *"làm thế nào ?"*

Trước hết phải nghĩ ngay tới : dãy số ở đây được định hình theo cấu trúc nào ? (cấu trúc dữ liệu) và được cài đặt trong máy theo cấu trúc nào ? (mà ta sẽ được gọi là : *cấu trúc lưu trữ*).

Thông thường nó được định hình và cài đặt theo cấu trúc vectơ (ở chương 2 sẽ nói rõ hơn).

Ở đây có hai vectơ : vectơ input và vectơ output. Vậy thì trong máy ta sẽ dùng hai vectơ để lưu trữ hay chỉ dùng một ?

Giả sử ta chỉ dùng 1, nghĩa là lúc đầu vectơ lưu trữ chứa dãy số cho, nhưng sau khi thực hiện giải thuật thì chính vectơ ấy cũng chứa dãy số đã được sắp xếp (để tiết kiệm bộ nhớ !).

Nếu thế thì công việc "đổi chỗ" sẽ được cụ thể thêm như sau :

– Hoán vị vị trí của nó (số bé nhất vừa được chọn) với vị trí của số ở đầu dãy chưa được sắp, sau đó gạt nó ra ngoài dãy chưa được sắp (tất nhiên lúc đó nó đã trở thành phần tử cuối của dãy đã được sắp).

Tới đây ta có thể diễn đạt sơ bộ giải thuật "sắp xếp" của ta như sau :

Procedure SELECTION–SORT (A,n) ;

{A là vectơ gồm n phần tử là các số cho}

1. {2 công việc được lặp lại (n–1) lần}

for i : = 1 **to** (n–1) **do begin**

2. Chọn số nhỏ nhất A[k] trong dãy các số :

A[i], A[i+1], ..., A[n]

3. Hoán vị giữa A[k] và A[i]

4. **return** **end** ;

Bây giờ ta đi sâu vào từng công việc :

• Làm thế nào để chọn được số nhỏ nhất trong dãy các số :

A[i], A[i+1],..., A[n] ?

Có thể tiến hành như sau : thoạt đầu ta cứ chọn A[i], sau đó so sánh các phần tử tiếp theo với nó, nếu phần tử nào nhỏ hơn thì lại thay phần tử đó vào, phần tử cuối cùng được thay chính là phần tử cần tìm.

Nhưng xét cho cùng : ta chỉ cần biết chỉ số k ứng với phần tử nhỏ nhất đó thì sẽ tìm được nó, vì vậy công việc "chọn" ở trên chỉ cần làm với chỉ số. Có thể diễn đạt như sau :

```
k : = i ; {coi phần tử đầu là nhỏ nhất lúc đó, và giữ lại chỉ số của nó}
    for j : = i + 1 to n do
        if A[j] < A[k] then k : = j
```

• Làm thế nào để thực hiện được việc hoán vị chỗ cho hai phần tử ?

Cách giải quyết ở đây giống như khi ta có 2 cốc khác nhau : một đựng rượu, một đựng nước ; mà ta lại muốn hoán vị 2 thứ chất lỏng này nghĩa là chuyển nước sang cốc đang đựng rượu và chuyển rượu sang cốc đang đựng nước.

Rõ ràng điều này chỉ có thể thực hiện được khi ta dùng tới một cốc thứ ba làm cốc trung chuyển.

Từ đó ta có thể diễn đạt việc hoán vị giữa A[k] và A[i] như sau :

```
LOC : = A[k] ; A[k] : = A[i] ; A[i] : = LOC ;
```

Tổng hợp những ghi nhận ở trên, ta đi tới một thủ tục, thể hiện giải thuật "sắp xếp" của ta, bằng ngôn ngữ tựa PASCAL như sau :

```
Procedure SELECTION–SORT (A, n) ;
1. for i : = 1 to (n – 1) do begin
2. k : = i ;
3. for j : = i +1 to n do
4. if A[j] < A[k] then k : = j ;
5. LOC : = A[k] ; A[k] : = A[i] ; A[i] : = LOC
                                    end ;
6. return
```

Chú ý : 1) Cách làm ở trên phản ảnh một phương pháp thiết kế giải thuật, gắn liền với lập trình được gọi là "phương pháp tinh chỉnh từng bước" (stepwise refinement).

2) Cách cài đặt một "cấu trúc dữ liệu" ở trong MTĐT có thể khác nhau vì vậy để phân biệt ta gọi cấu trúc cài đặt trong máy của một "cấu trúc dữ liệu" là 'cấu trúc lưu trữ".

Như vậy nghĩa là : một cấu trúc dữ liệu có thể được cài đặt trong máy bởi các cấu trúc lưu trữ khác nhau.

Cũng như, ta sẽ thấy, một cấu trúc lưu trữ có thể biểu diễn được cho nhiều cấu trúc dữ liệu khác nhau.

3) Giải thuật sắp xếp ở trên thể hiện một kĩ thuật sắp xếp cơ bản được gọi là "sắp xếp kiểu lựa chọn" (vì thế thủ tục có tên SELECTION–SORT)

4) Thực ra việc đổi chỗ chỉ cần làm khi k ≠ j, nên câu lệnh 5 có thể viết :

```
if k ≠ i then begin
LOC : = A[k] ; A[k] : = A[i] ; A[i] : = LOC
                end ;
```

1.4. ĐÁNH GIÁ GIẢI THUẬT

1.4.1. Đặt vấn đề

Đối với một bài toán thường không phải chỉ có một giải thuật để giải nó mà có thể có nhiều giải thuật khác nhau (ứng với các cấu trúc dữ liệu hoặc cấu trúc lưu trữ khác nhau).

Từ đó, xuất hiện một mong muốn là làm sao tìm được giải thuật "tốt nhất", nhưng "tốt" nghĩa là thế nào ?

Khi một giải thuật được thực hiện thường nó liên quan đến hai yếu tố :

– *Không gian nhớ* cần thiết cho những cấu trúc lưu trữ.

– *Thời gian* cần thiết để thực hiện.

Việc đánh giá được thời gian thực hiện và không gian nhớ cần thiết của một giải thuật sẽ cho ta cơ sở để xác định được giải thuật nào là "tốt hơn". Tuy nhiên hai yếu tố "không gian" và "thời gian" ứng với giải thuật lại hay mâu thuẫn : "tốt" về thời gian, nghĩa là thực hiện nhanh, thường lại kéo theo "không tốt" về không gian, nghĩa là tốn nhiều bộ nhớ, và ngược lại. Vì vậy trong thực tế, đối với từng loại bài toán, một trong hai yếu tố đó sẽ được coi trọng hơn.

Thông thường, thời gian thực hiện giải thuật vẫn được chú ý hơn. Vì vậy, sau đây ta sẽ xét tới việc đánh giá thời gian thực hiện giải thuật .

Thời gian thực hiện một giải thuật chịu ảnh hưởng của nhiều yếu tố. Như ta đã biết : các kiểu lệnh và thời gian thực hiện các lệnh của các loại máy tính thường khác nhau. Hơn nữa ngôn ngữ lập trình và chất lượng của chương trình dịch cũng là các yếu tố liên quan tới thời gian thực hiện giải thuật. Vì vậy ta không thể tính thời gian này bằng phút, bằng giây... như cách đo thời gian thông thường để rồi so sánh với nhau.

Cùng một giải thuật, nhưng thực hiện trên hai loại máy khác nhau, với ngôn ngữ lập trình và chương trình dịch khác nhau sẽ đưa tới chi phí về thời gian tính theo phút, theo giây... khác nhau.

Vậy thì dựa vào đâu để có thể nói rằng : giải thuật này "nhanh hơn" giải thuật kia ?

Trước hết ta thấy : Thời gian thực hiện giải thuật thường phụ thuộc vào kích thước của bộ dữ liệu (nói gọn là *kích thước dữ liệu*). Ví dụ :

Sắp xếp một dãy n số, thì kích thước dữ liệu là n ; n càng lớn thì thời gian sắp xếp càng lâu. Do đó người ta tìm cách biểu diễn thời gian thực hiện giải thuật bằng một hàm số của kích thước n : T(n) (việc xác định kích thước của dữ liệu tuỳ thuộc vào từng bài toán cụ thể).

Rõ ràng là T(n) độc lập với các yếu tố khách quan đã nêu ở trên. Với cách tiếp cận này, cùng một bài toán, nếu giải thuật A_1 có thời gian thực hiện là $T_1(n) = 8n$, và một giải thuật A_2, có thời gian thực hiện là $T_2(n) = 2n^2$ thì khi n đủ lớn ta thấy $T_1(n) \leq T_2(n)$ (ở đây chỉ cần $n \geq 4$ là $2n^2 \geq 8n$) và n, càng lớn thì sự chênh lệch càng rõ. Như vậy lúc đó ta có thể nói :

Khi n đủ lớn thì giải thuật A_1 "nhanh hơn" giải thuật A_2.

Trong thực tế, với tốc độ tính toán của MTĐT như hiện nay, thì việc so sánh thời gian thực hiện giải thuật chỉ đặt ra khi n khá lớn thôi (lúc đó độ chênh lệch mới đáng kể).

Vấn đề đặt ra bây giờ là : Làm thế nào để xác định được T(n) ?

– Trước hết ta hãy xét một ví dụ :

Giải thuật tính giá trị trung bình của n số :

Program TB

{các số ở đây được coi như n giá trị khác nhau của X ; M sẽ lưu giữ giá trị trung bình sau khi được tính}.

1. **Read** (n) ;

2. S : = 0 ;

3. i : = 1 ;

4. **While** i ≤ n **do begin**

5. **Read** (X) ;

6. S : = S + X ;

7. i : = i + 1

end ;

8. M : = S/n ; **Write** (M) ;

9. **return**

Ta thấy các lệnh 1,2, 3, 8 được thực hiện 1 lần. Các lệnh 5, 6, 7, tạo ra thân của vòng lặp được thực hiện mỗi lệnh n lần. Lệnh 4, kiểm tra sự lặp lại được thực hiện (n + 1) lần. Tổng cộng số lệnh thực hiện là 4n + 5. Dù thực hiện trên máy nào thì số lệnh này vẫn như vậy, và nó ảnh hưởng tới thời gian thực hiện giải thuật.

Do đó ta coi : T(n) = 4n + 5

Khi giá trị của n tăng thì giá trị của T(n) cũng tăng một cách tuyến tính. Ta nói : T(n) có độ lớn bậc n. Điều này thường được kí hiệu theo "kí pháp chữ O lớn" là : T(n) = O(n)

Một cách tổng quát : Thời gian T(n) của một giải thuật được gọi là có độ lớn bậc f(n), kí hiệu bởi : T(n) = O(f(n)), nếu tồn tại các số dương C và n_0 sao cho :

$$T(n) \leq Cf(n), \forall\ n \geq n_0$$

Lúc đó người ta cũng nói : *độ phức tạp* về thời gian của giải thuật này là O(f(n)).

Với giải thuật "tính giá trị trung bình" ở trên độ phức tạp của nó là O(n) vì :

$$T(n) = 4n + 5$$

mà 4n + 5 ≤ 5n, với ∀ n ≥ 5 ;

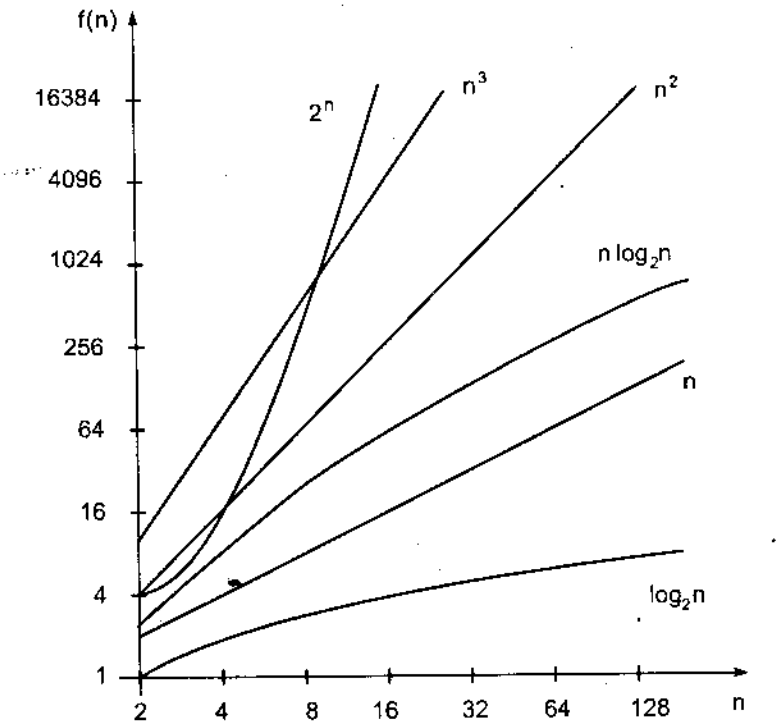
Như vậy chỉ cần chọn f(n) = n, n_0 = 5, C = 5 là thỏa mãn.

Thường người ta chọn f(n) là các hàm đơn giản để biểu diễn độ phức tạp của một giải thuật.

Sau đây là một số hàm thông dụng :

Log_2n ; n ; $nlog_2n$; n^2 ; n^3 ; 2^n

Chúng được sắp xếp theo thứ tự tăng dần. Ta có thể thấy rõ sự tăng trưởng giá trị của các hàm này theo giá trị của n, qua đồ thị dưới đây :



Chú ý : 1) Ta cũng thấy thêm là khi biểu diễn T(n) dưới dạng O(f(n) thì hằng số nhân không đóng vai trò quan trọng. Như với 2 giải thuật A_1 và A_2 nêu trên thì có thể viết :

$$T_1(n) = O(n) ; T_2(n) = O(n^2)$$

2) Với T(n) là một đa thức, có dạng :

$$T(n) = a_k n^k + a_{k-1} n^{k-1} + ...+ a_1 n + a_0$$

thì cũng chứng minh được, khi n đủ lớn,

$$T(n) = O(n^k)$$

Nghĩa là khi n đủ lớn, thì số hạng với mũ lớn nhất sẽ được coi trọng, khi đánh giá thời gian thực hiện giải thuật.

3) Từ những nhận xét trên, khi xác định độ phức tạp của giải thuật, theo kí pháp chữ O, người ta chỉ cần chú ý tới phép toán nào đó mà số lần thực hiện nó phụ thuộc vào n và không thua kém các phép khác ; người ta gọi là "phép toán tích cực" (active operation) và thời gian thực hiện giải thuật sẽ được đánh giá về bậc theo số lần thực hiện phép này.

Như trong giải thuật "tính giá trị trung bình" ở trên : có thể coi phép so sánh $i \leq n$ trong câu lệnh **while** làm phép toán tích cực. Số lần thực hiện nó là n + 1, từ đó suy ra :

$$T(n) = O(n)$$

Ta sẽ xét thêm điều này, qua giải thuật SELECTION–SORT nêu ở mục 1.3.

Ở đây có thể coi phép so sánh A[j] < A[k] là "phép tích cực". Ta thấy :

với i = 1 phép này được thực hiện (n – 1) lần

i = 2 phép này được thực hiện (n – 2) lần

...

i = (n – 1) phép này được thực hiện 1 lần

Vậy tổng số lần thực hiện nó là :

$$1 + 2 + ... + (n - 1) = \frac{(n-1)n}{2}$$

$$= \frac{n^2}{2} - \frac{n}{2}$$

Do đó suy ra $T(n) = O(n^2)$

Nếu xét lại một cách tỉ mỉ thì :

Lệnh gán giá trị cho i ở bước 1 được thực hiện (n – 1) lần

Các lệnh ở bước 2, 5 mỗi lệnh được thực hiện (n – 1) lần

Lệnh gán giá trị cho j ở bước 3 : được thực hiện :

$$(n - 1) + (n - 2) + ... + 1 = \frac{n(n-1)}{2}$$

Lệnh so sánh A[j] < A[k] cũng được thực hiện $\frac{n(n-1)}{2}$ lần (như đã tính).

Riêng lệnh gán k : = j ở bước 4, nhiều nhất cũng chỉ thực hiện (n – 1) lần (chỉ khi A[j] <A[k] có giá trị *true* thì lệnh này mới thực hiện).

Vì vậy có thể coi :

$$T(n) \le (n-1) + 4(n-1) + \frac{n(n-1)}{2} + \frac{n(n-1)}{2} + n - 1$$

$$\le n^2 + 5n - 6$$

Vì $n \le n^2 \quad \forall n \ge 0$, ta có :

$$n^2 + 5n - 6 \le n^2 + 5n^2 = 6n^2$$

Do đó $T(n) \le 6n^2 \ \forall n \ge 0$.

Vậy chọn $f(n) = n^2$, $n_0 = 0$ và $C = 6$ là ta có thể viết : $T(n) = O(n^2)$.

Kết quả này trùng khớp với cách tính dựa vào phép tích cực nêu ở trên cũng như nhận xét ở phần chú ý 1.

1.4.2. Thời gian trung bình

Có nhiều trường hợp thời gian thực hiện giải thuật T(n) không những phụ thuộc vào kích thước của dữ liệu mà còn phụ thuộc vào tình trạng của dữ liệu nữa.

Trở lại bài toán tìm kiếm một số X trong dãy n số a_1 ; a_2 ;.... ; a_n, theo phương pháp tìm kiếm tuần tự, ta thấy ngay : Với phép tính tích cực là phép so sánh thì : Nếu $X = a_1$ ta chỉ cần một phép so sánh.

Nếu $X = a_n$ hoặc không có a[i] nào ($1 \le i \le n$) có giá trị bằng X thì cần tới n phép so sánh.

Như vậy là tốt nhất thì $T(n) = O(1)$ (ta kí hiệu là $T_t(n) = O(1)$.

Còn xấu nhất thì $T(n) = O(n)$ (ta kí hiệu là $T_x(n) = O(n)$.

Vậy thì, tất nhiên phải đặt ra vấn đề : thời gian trung bình sẽ là bao nhiêu ? ($T_{tb}(n) = ?$)

Việc tính giá trị trung bình của thời gian thực hiện giải thuật thường phức tạp vì nó liên quan đến tính ngẫu nhiên của các sự kiện. Nó đòi hỏi phải sử dụng tới phép tính xác suất và thống kê nên ta sẽ không tìm hiểu sâu ở đây.

Với giải thuật tìm kiếm tuần tự, người ta chứng minh được rằng : $T_{tb}(n) = O(n)$.

Trong một số trường hợp, khi không biết $T_{tb}(n)$ thì người ta có thể dùng $T_x(n)$ để ước lượng.

Tuy nhiên, cần thấy rằng, một cách tổng quát thì $T_x(n)$ chỉ cho ta cận trên tối đa của thời gian thực hiện giải thuật thôi, nói nôm na ra thì điều đó có nghĩa là :"lấy già ra thì thời gian thực hiện giải thuật có bậc như vậy" (nếu bậc đó không cao thì giải thuật đó cũng "tốt") còn thời gian trung bình thì có thể có bậc như thế, nhưng cũng có thể có bậc thấp hơn !

Dù sao thì với kí pháp chữ O lớn, ta cũng biết được độ đo gần đúng của thời gian thực hiện giải thuật khi kích thước dữ liệu đủ lớn. Điều đó cũng giúp ta có cơ sở đánh giá một cách tương đối về thời gian này, đối với các giải thuật khác nhau.

1.5. GIẢI THUẬT ĐỆ QUY

1.5.1. Định nghĩa

Đệ quy là một khái niệm có vai trò rất quan trọng trong tin học.

Một đối tượng gọi là đệ quy nếu nó bao gồm chính nó như một bộ phận.

Một hàm gọi là đệ quy nếu trong định nghĩa của nó lại có dạng của chính nó.

Một ví dụ, khá quen thuộc về hàm đệ quy là hàm tính giai thừa của một số nguyên không âm : n! với quy ước 0! = 1 thì hàm này sẽ được định nghĩa như sau :

1. Nếu n = 0 thì n! = 1
2. Nếu n > 0 thì n! = n(n – 1)!

Như vậy trong định nghĩa của n! lại có (n–1)! đó chính là tính đệ quy. Ta có cảm giác như nó luẩn quẩn, nhưng thực ra lại rất rõ ràng. Ví dụ : ta muốn tính 4!, theo định nghĩa ta có :

4! = 4.3!
↓
3! = 3.2!
↓
2! = 2.1!
↓
1! = 1.0!
nhưng 0! = 1

Vậy thì 4! = 4.3. 2.1.1 = 24

Một giải thuật thể hiện được cách tính giá trị của một hàm theo định nghĩa đệ quy, hay nói một cách tổng quát : thể hiện được cách xử lí đệ quy để giải quyết một bài toán, thì được gọi là *giải thuật đệ quy*. Nếu giải thuật ấy được viết dưới dạng một thủ tục thì thủ tục ấy được gọi là *thủ tục đệ quy*.

Sau đây ta sẽ xét một số ví dụ về thủ tục đệ quy.

1.5.2. Ví dụ về thủ tục đệ quy

1. Hàm tính n!

Dựa theo định nghĩa đã nêu ở trên, giải thuật đệ quy n! được viết dưới dạng thủ tục hàm như sau :

```
Function FACT (n) ;
1. if n = 0  then FACT : = 1
                 else FACT : = n * FACT (n–1) ;
2. return
```

Như vậy ta sẽ thấy thủ tục này được viết dưới dạng tương tự như định nghĩa của n !. Tính đệ quy của thủ tục này được thể hiện qua 2 đặc điểm :

a) Có một số trường hợp đặc biệt, mà ta sẽ gọi là *trường hợp suy biến*, ứng với một "tiêu chuẩn gốc" (ở đây là n = 0), thì việc xử lí được thực hiện cụ thể theo một cách riêng.

b) Còn các trường hợp khác, trong xử lí đều có sự tham chiếu đến chính nó (như ở đây là gọi đến chính nó : FACT (n – 1). Tuy nhiên, phải chú ý là khi có sự tham chiếu đến chính nó thì nó lại tiến gần hơn đến trường hợp suy biến (ở đây là kích thước (n – 1) sẽ nhỏ hơn n và gần với 0 hơn n).

2. Dãy số Fibonacci

Dãy số Fibonacci là dãy số có dạng như sau :

1, 1, 2, 3, 5, 8, 13, 21, 34, 55, ...

Với hai số đầu là 1 và 1 thì mỗi số sau sẽ là tổng của hai số đứng trước. Các số này được coi là giá trị của một hàm Fib với đối số là số nguyên dương n mà ta gọi là hàm *Fibonacci*. Ta có thể định nghĩa Fib(n) như sau :

1. Nếu n = 1 hoặc n = 2 thì Fib(n) = 1

2. Nếu n > 2 thì Fib(n) = Fib(n–1) + Fib(n–2)

Từ đó ta có thể viết giải thuật tính giá trị của Fib(n) dưới dạng thủ tục đệ quy như sau :

Function FIB(n) ;

1. **if** n ≤ 2 **then** FIB : = 1

   else FIB : = FIB(n – 1) + FIB (n – 2)

2. **return**

Về đặc điểm ta cũng thấy thủ tục này vẫn có 2 đặc điểm như đã nêu. Tuy nhiên cần chú ý rằng : tiêu chuẩn gốc ở trường hợp suy biến là ứng với 2 giá trị đầu tiên, và việc tính các giá trị sau là ứng với hai lần thủ tục gọi lại chính nó (FIB (n – 1) và FIB (n – 2)).

Hai thủ tục mà ta nêu trên đều ứng với giải thuật tính giá trị hàm, mà định nghĩa đệ quy của nó xác định được khá dễ dàng và giải thuật hầu như "được phỏng theo" định nghĩa ! Tuy nhiên không phải các giải thuật (hay thủ tục) đệ quy chỉ liên quan tới việc tính giá trị hàm, mà trong nhiều bài toán khác, ta có thể tìm ra cách giải đệ quy, nếu ta biết đưa bài toán đó đến một bài toán con tương tự như nó (tất nhiên điều này không phải dễ dàng và không phải lúc nào cũng làm được !).

Sau đây ta sẽ xét một bài toán khác mà cách "xử lí đệ quy" lại tỏ ra rất dễ hiểu.

3. Bài toán "Tháp Hà Nội"

Đây là một bài toán mang tính chất một trò chơi, với nội dung như sau :

– Có n đĩa, kích thước nhỏ dần, đĩa có lỗ ở giữa (như đĩa CD). Có thể xếp chúng chồng lên nhau xuyên qua một cọc, to dưới nhỏ trên, để cuối cùng có một chồng đĩa giống như hình cái tháp (như dạng tháp rùa ở Hồ Gươm, Hà Nội).

– Có 3 cọc A, B, C. Hiện n đĩa đang xếp theo hình tháp ở cọc A, yêu cầu đặt ra là :

Chuyển chồng đĩa từ cọc A sang cọc C, theo những điều kiện sau :

1. Mỗi lần chỉ được chuyển một đĩa

2. Không khi nào có tình huống đĩa to ở trên, đĩa nhỏ ở dưới.

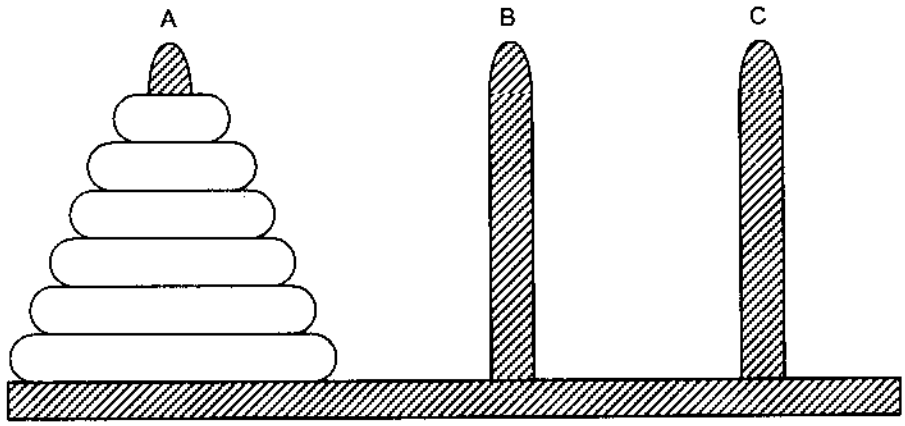
3. Được phép sử dụng một cọc làm cọc trung chuyển, chẳng hạn khi chuyển đĩa từ cọc A sang cọc C thì cọc B được dùng làm cọc trung chuyển.

Hình 1.1 ứng với dạng ban đầu của bài toán, với n = 6.

Trước hết, ta hãy xét vài trường hợp đơn giản :

a) Trường hợp n = 1 : chỉ cần 1 phép chuyển.

– Chuyển đĩa đang ở A sang C (kí hiệu là A → C)



HÌNH 1.1

b) Trường hợp n = 2 : phải thực hiện 3 phép chuyển

– Chuyển đĩa thứ nhất từ cọc A sang cọc B : A → B

– Chuyển đĩa thứ hai từ cọc A sang cọc C : A → C

– Chuyển đĩa thứ nhất từ cọc B sang cọc C : B → C

c) Trường hợp n > 2.

Ta thấy nếu coi (n–1) đĩa ở trên đóng vai trò như đĩa thứ nhất thì có thể hình dung như đang có 2 đĩa ở cọc A. Nếu thế thì phỏng theo trường hợp 2 đĩa ta có thể đi tới giải thuật như sau :

– Chuyển (n – 1) đĩa trên từ A sang B

– Chuyển đĩa thứ n từ A sang C

– Chuyển (n – 1) đĩa từ B sang C

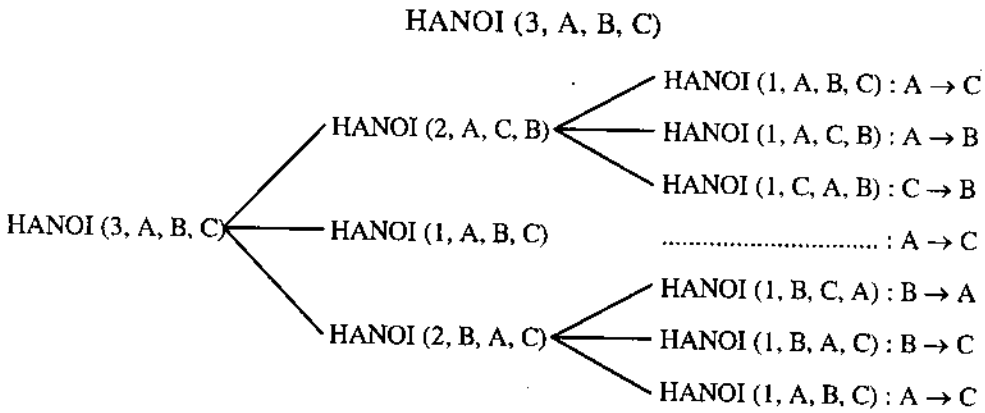
Lược đồ 3 bước này đã đưa bài toán "tháp Hà Nội" ứng với n đĩa, đến bài toán ứng với (n – 1) đĩa và ở mức này thì lại dẫn tới bài toán với (n – 2) đĩa... và cuối cùng sẽ dẫn tới bài toán ứng với 1 đĩa nghĩa là tới bài toán đơn giản : chuyển 1 đĩa từ cọc này sang cọc kia.

Vậy thì cách giải này đã mang tính chất đệ quy và giải thuật tương ứng sẽ được thể hiện qua thủ tục đệ quy như sau :

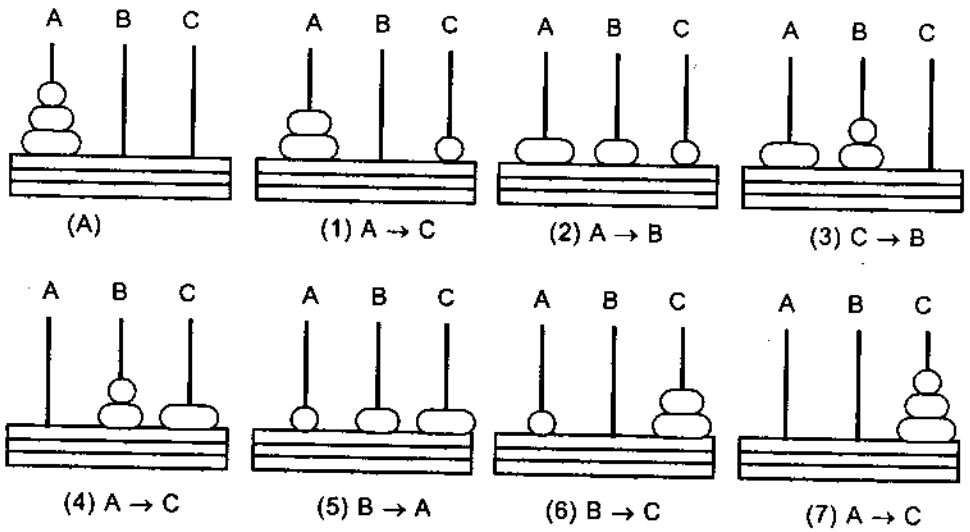
Procedure HANOI (n, A, B, C) ;

1. **if** n = 1 **then** chuyển đĩa từ A sang C
2. **else** **begin** **call** HANOI (n–1, A, C, B) ;
   call HANOI (1, A, B, C) ;
   call HANOI (n–1, B, A, C) ;
   end
3. **return**

Sau đây là sơ đồ thực hiện thủ tục



Hình 1.2 mô tả 7 bước thực hiện chuyển đĩa như trên :



HÌNH 1.2

1.5.3. Chú ý

1) Với giải thuật HANOI (n, A,B, C) người ta đánh giá thời gian thực hiện giải thuật dựa theo số lần chuyển đĩa.

Người ta chứng minh được rằng : số lần chuyển đĩa này là $C(n) = 2^n - 1$. Như vậy có thể suy ra $T(n) = O(2^n)$, điều đó cho thấy rằng : chi phí về thời gian thực hiện giải thuật là rất cao. Có thể thấy ngay : với n = 20 thì số lần chuyển đĩa đã lên tới hàng triệu, n = 30 thì đã tăng lên đến hàng tỉ.

Người ta gọi các hàm $\geq 2^n$ là các hàm loại mũ, còn các hàm như $\log_2 n$, n, $n\log_2 n$, n^2, n^3 được gọi là các hàm loại đa thức.

Nếu thời gian thực hiện giải thuật có bậc thuộc bậc hàm loại đa thức thì thường chấp nhận được, còn nếu thuộc bậc hàm loại mũ thì trong thực tế, với tốc độ xử lí như các máy tính hiện nay cũng không thể thực hiện được với một giá trị n đủ lớn nào đó (mặc dầu giải thuật xử lí rất đúng đắn).

2) Giải thuật đệ quy thường "ngắn gọn" và cách viết khá đơn giản, đó là điều thuận lợi cho người lập trình, nếu như họ biết cách khai thác được tính đệ quy trong cách giải. Tuy nhiên điều đó không có nghĩa là chúng sẽ thực hiện nhanh. Với hàm giai thừa hoặc hàm Fibonacci nêu trên, ta có thể lập được giải thuật không đệ quy (dùng phép lặp để tính) mà thời gian thực hiện sẽ nhanh hơn.

Vì vậy, chỉ nên coi đệ quy là một công cụ để giải bài toán. Đối với người làm tin học, cũng nên làm quen với cách tiếp cận đệ quy, khi thiết lập giải thuật. Còn đánh giá về thời gian thực hiện chúng cũng như đối với các giải thuật khác : Có thể có giải thuật đệ quy "không tốt", nhưng cũng có thể có giải thuật đệ quy "tốt" hơn (chẳng hạn giải thuật QUICK-SORT ở chương 2 là một giải thuật sắp xếp khá tốt).

3) Các ngôn ngữ như PASCAL, C v. v. đều cho phép viết thủ tục dưới dạng đệ quy. Điều đó cũng có ý nghĩa là chương trình dịch của các ngôn ngữ này sẽ đảm nhiệm việc chuyển thủ tục đó sang một thủ tục tương đương mà không đệ quy, (gọi là "khử đệ quy") vì trong máy tính điện tử không hề có "phép tính đệ quy".

Có thể nói rằng một ngôn ngữ (tất nhiên bao hàm cả chương trình dịch ngôn ngữ đó) cho phép viết đệ quy, nghĩa là nó nhận khó khăn về phía mình để tạo thuận lợi cho người sử dụng, vì vậy nếu người sử dụng không tiếp cận được với những thuận lợi đó thì cũng là điều đáng tiếc.

Câu hỏi và bài tập

1.1. Cấu trúc dữ liệu và cấu trúc lưu trữ khác nhau ở chỗ nào ?

1.2. Hãy nêu một vài cấu trúc dữ liệu của ngôn ngữ lập trình mà anh (chị) biết .

1.3. Các cấu trúc dữ liệu tiền định của một ngôn ngữ lập trình có đủ đáp ứng mọi yêu cầu về tổ chức dữ liệu không ?

1.4. Hãy nêu một giải thuật mà độ phức tạp về thời gian của nó là O(1).

1.5. Có người nói : "Phép đệ quy phản ánh chiến thuật "chia để trị" trong cách giải bài toán". Điều đó có đúng không ?

1.6. Hãy lập bảng so sánh giá trị hai hàm $f(n) = n^2$ và $g(n) = 2^n/4$ tương ứng với một số giá trị của n. Xác định xem từ giá trị nào của n thì $g(n) \geq f(n)$. Kể từ giá trị đó trở đi thì có khi nào $g(n) < f(n)$ không ? Có phải lúc nào g(n) cũng lớn hơn f(n) không ?

1.7. Kí pháp "chữ O lớn" nào tốt nhất để biểu diễn thời gian thực hiện sau đây :

a) $T(n) = n^3 + 100n\log_2 n + 5000$; b) $T(n) = 2^n + n^{99} + 7$

c) $T(n) = n*(3 + n) - 7n$; d) $T(n) = \dfrac{(n+1)*\log_2(n+1) - (n+1) + 1}{n}$

1.8. Với mỗi đoạn giải thuật dưới đây, hãy dùng kí pháp "chữ O lớn" tốt nhất để biểu diễn thời gian thực hiện.

a)
```
for i : = 1 to n do
        for j : = 1 to n do
                A[i,j] : = B[i,j] + C[i,j]
```

b)
```
s : = 0 ;
for i : = 1 to n do begin
                read (x) ;
                s : = s + x
                end
```

c)
```
for i : = 1 to n do
        for j : = 1 to n do begin
                        C[i,j] : = 0 ;
                for k : = 1 to n do
                C[i,j] : = C[i,j] + A[i,k] * B[k,j]
                                end
```

d)
```
j : = n ;
repeat
j : = j/2
until j ≤ 1
```

1.9. Giả sử a và b là những số nguyên dương. Q là hàm số của a, b, được định nghĩa như sau :

$$Q(a,b) = \begin{cases} 0 \text{ nếu } a < b \\ Q(a - b, b) + 1 \text{ nếu } a \geq b \end{cases}$$

Hãy tính Q(2,3) và Q(14,3)

1.10. Hàm Ackermann là hàm hai đối số với giá trị của đối số là số nguyên không âm. Nó được định nghĩa như sau :

$$\text{Acker } (m,n) = \begin{cases} n + 1 \text{ nếu } m = 0 \\ \text{Acker } (m - 1, 1) \text{ nếu } m \neq 0, n = 0 \\ \text{Acker } (m - 1, \text{Acker } (m, n - 1)) \text{ nếu } m \neq 0 \text{ và } n \neq 0 \end{cases}$$

a) Hãy xác định xem ở đây "tiêu chuẩn gốc" (ứng với trường hợp suy biến) là gì ?

b) Hãy tính Acker (1, 3).

1.11. Cho biết số Fibonacci $F_{11} = 89$ và $F_{12} = 144$

a) Hãy tính F_{16}

b) Viết một thủ tục không đệ quy (dùng phép lặp) để tính và in ra n số Fibonacci đầu tiên.

1.12. Giải thuật tính ước số chung lớn nhất của 2 số p và q (p > q) được mô tả như sau (giải thuật Euclide)

Gọi r là số dư trong phép chia p cho q :

– Nếu r = 0 thì q là ước số chung lớn nhất

– Nếu r ≠ 0 thì gán cho p giá trị của q, gán cho q giá trị của r rồi lặp lại quá trình trên.

a) Hãy lập bảng ghi nhận các giá trị của p, q, r trong quá trình thực hiện tính ước số chung lớn nhất của 2 số : 1260 và 198.

b) Hãy nêu lên tính đệ quy trong cách tính này từ đó xây dựng một cách tính đệ quy cho hàm tính ước số chung lớn nhất

USCLN (p,q)

c) Viết một giải thuật đệ quy và một giải thuật không đệ quy (dùng phép lặp) để tính ước số chung lớn nhất của p,q.

Chương 2. CẤU TRÚC MẢNG (ARRAY)

Cấu trúc dữ liệu đầu tiên mà ta nói tới là cấu trúc *mảng*, đây là một cấu trúc rất quen thuộc, nó có mặt ở hầu hết các ngôn ngữ lập trình.

2.1. ĐỊNH NGHĨA

Mảng là một tập hợp có thứ tự, bao gồm một số xác định n phần tử (n được gọi là *độ dài* hay *kích thước* của mảng). Ngoài giá trị, mỗi phần tử của mảng còn được đặc trưng bởi chỉ số (index), thể hiện thứ tự của phần tử đó trong mảng. Các giá trị của phần tử mảng đều cùng một loại.

Đối với mảng thường có các phép toán :

– Tạo lập một mảng.

– Duyệt qua các phần tử của mảng.

– Tìm kiếm một phần tử của mảng.

– Sắp xếp các phần tử trong mảng theo một thứ tự ấn định v. v....

Vì số phần tử của mảng là cố định, nên không có phép bổ sung phần tử mới vào mảng hoặc loại bỏ một phần tử ra khỏi mảng.

Vectơ là mảng một chiều, mỗi phần tử của nó ứng với một chỉ số. Chẳng hạn : phần tử của vectơ A, kí hiệu là A_i hoặc A[i] với i là chỉ số.

Ma trận là mảng hai chiều, mỗi phần tử của nó ứng với 2 chỉ số, ví dụ : phần tử của ma trận B, kí hiệu là B_{ij} hoặc B[i,j] với i gọi là chỉ số hàng, j gọi là chỉ số cột.

Tương tự người ta cũng mở rộng : mảng ba chiều, mảng bốn chiều,.... mảng n chiều.

2.2. CẤU TRÚC LƯU TRỮ CỦA MẢNG

2.2.1. Khái quát về cách lưu trữ

Một cách đơn giản, có thể hình dung bộ nhớ của máy tính điện tử (MTĐT) là một dãy các phần tử nhớ cơ sở được đánh số kế tiếp nhau (kể từ số 0). Số thứ tự đó được gọi là *địa chỉ*, một phần tử nhớ cơ sở, có địa chỉ được gọi là một *từ máy*. Một phần tử dữ liệu có thể được lưu trữ trong máy bởi một *ô nhớ*

bao gồm một hoặc nhiều từ máy. Việc truy cập vào ô nhớ đó sẽ được xác định bởi địa chỉ của từ máy đầu tiên tạo nên ô nhớ đó. Thường có hai cách để xác định được địa chỉ.

Cách thứ nhất là dựa vào những đặc tả của việc lưu trữ dữ liệu để tính trực tiếp ra địa chỉ. Địa chỉ loại này gọi là *địa chỉ được tính* (computer address). Cách này thường hay được sử dụng trong chương trình dịch của các ngôn ngữ lập trình để tính địa chỉ các phần tử của mảng, tính địa chỉ các lệnh thực hiện tiếp theo v. v...

Cách thứ hai là lưu trữ các địa chỉ cần thiết ở một chỗ quy định, khi cần xác định sẽ lấy từ đó ra. Loại địa chỉ này được gọi là *con trỏ* (pointer) hoặc *mối nối* (link). Địa chỉ quay lui của chương trình con để quay trở về chỗ có lời gọi trong chương trình chính, khi kết thúc việc thực hiện chương trình con đó, chính là loại địa chỉ này.

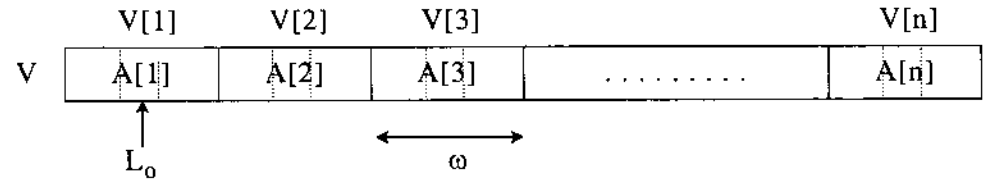
Cũng có một số cấu trúc lưu trữ sử dụng phối hợp cả hai cách xác định địa chỉ nói trên.

2.2.2. Lưu trữ kế tiếp đối với mảng

Thông thường mảng được lưu trữ trong máy dưới dạng một vectơ, mà người ta gọi đó là *vectơ lưu trữ*. Đó là một dãy các từ máy kế tiếp nhau (vì vậy người ta gọi là cách *lưu trữ kế tiếp* – sequential storage allocation).

Giả sử, ta xét việc lưu trữ kế tiếp đối với mảng một chiều, hay một vectơ A, mà các phần tử của nó là A[i] với $1 \le i \le n$. Nếu mỗi phần tử của vectơ được lưu trữ trong một ô nhớ gồm có 1 từ máy thì để lưu trữ vectơ A, phải dành ra trong bộ nhớ n từ máy kế tiếp nhau, đó chính là n phần tử của vectơ lưu trữ V : phần tử V[i] của vectơ V sẽ chứa một phần tử A[i] của vectơ đang xét.

Nếu mỗi phần tử của vectơ lưu trữ V (mỗi ô nhớ của V) phải gồm ω từ máy mới đủ chứa được một phần tử A[i] thì lúc đó V phải bao gồm $n*\omega$ từ máy kế tiếp. Địa chỉ của mỗi ô nhớ, nghĩa là mỗi phần tử nhớ V[i], bây giờ là địa chỉ của từ máy đầu tiên của ô nhớ đó. Ví dụ : nếu ω = 3 mà địa chỉ của V[1] là 1000 thì địa chỉ của V[2] là 1003, của V[3] là 1006.



HÌNH 2.1

Địa chỉ của V[1] được gọi là *địa chỉ gốc* (base address), kí hiệu là L_0.

Như vậy việc xác định địa chỉ của V[i], hay nói một cách khác : việc xác định địa chỉ của A[i] sẽ được tính ra theo công thức sau :

$$LOC(A[i]) = L_0 + \omega * (i-1)$$

(dấu * ở đây biểu diễn phép nhân)

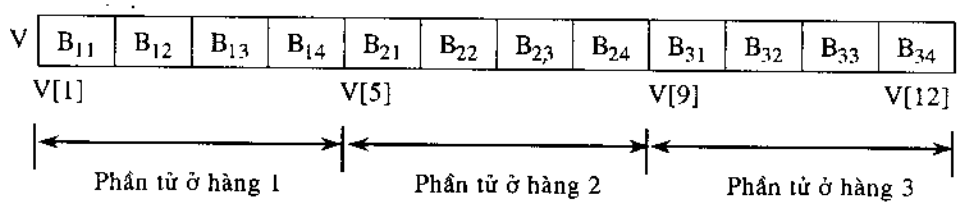
Trong ngôn ngữ như PASCAL, cận dưới của chỉ số không nhất thiết phải là 1, mà có thể là một số nguyên b nào đó. Khi ấy địa chỉ của A[i] được tính bởi :

$$LOC(A[i]) = L_0 + \omega * (i-b)$$

Đối với mảng 2 chiều, hay ma trận, việc lưu trữ các phần tử cũng được thực hiện bởi một vectơ lưu trữ như trên.

Gọi B là một ma trận có m hàng, n cột, B sẽ được lưu trữ trong bộ nhớ bởi vectơ lưu trữ V bao gồm m*n*ω từ máy (mỗi phần tử của V gồm ω từ máy).

Với ngôn ngữ PASCAL, nếu giả sử B có 3 hàng, 4 cột (m = 3, n = 4) thì các phần tử của nó sẽ được lưu trữ như hình sau :



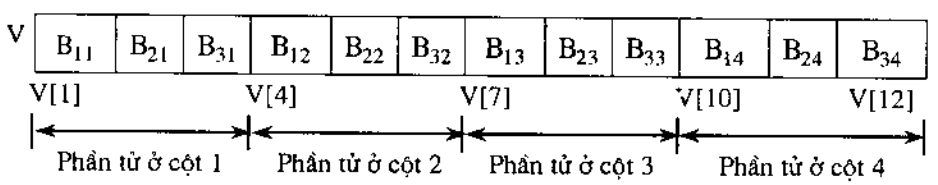
HÌNH 2.2

Như vậy nghĩa là : các phần tử của ma trận B sẽ được lưu trữ theo hàng, hết hàng này đến hàng khác.

Cách lưu trữ này được gọi là : *lưu trữ theo thứ tự ưu tiên hàng* (row–major order)

Cũng còn có một cách khác, đó là : *lưu trữ theo thứ tự ưu tiên cột* (column major order). Các phần tử của ma trận sẽ được lưu trữ theo cột, hết cột này đến cột khác.

Với ma trận B, 3 hàng 4 cột như trên thì các phần tử sẽ được lưu trữ bởi vectơ lưu trữ V theo như hình 2.3 :



HÌNH 2.3

Việc xây dựng các công thức tính địa chỉ cũng được tiến hành tương tự.

Nếu ma trận B có m hàng, n cột và mỗi phần tử của vectơ lưu trữ V gồm ω từ máy, thì địa chỉ của B[i,j] với $1 \le i,j \le n$:

Theo thứ tự ưu tiên hàng sẽ được tính bởi :

$$LOC(B[i,j]) = L_0 + [(i - 1) * n + (j - 1)] * \omega$$

Theo thứ tự ưu tiên cột sẽ được tính bởi :

$$LOC(B[i,j]) = L_0 + [(j - 1) * m + (i - 1)] * \omega$$

Trường hợp $b_1 \le i \le u_1$, $b_2 \le j \le u_2$ thì mỗi hàng sẽ có $(u_2 - b_2 + 1)$ phần tử. Khi đó công thức tính địa chỉ, chẳng hạn : theo thứ tự ưu tiên hàng, sẽ là :

$$LOC(B[i,j]) = L_0 + [(i - b_1) * (u_2 - b_2 + 1) + (j - b_2)] * \omega$$

Người ta cũng mở rộng cách lưu trữ tương tự đối với mảng nhiều chiều.

Chú ý : 1) Khi mảng được lưu trữ kế tiếp thì việc truy cập vào một phần tử của mảng được thực hiện một cách trực tiếp (*truy cập trực tiếp* – direct access) thông qua "địa chỉ được tính", nên tốc độ truy cập nhanh và đồng đều đối với mọi phần tử.

2) Đối với người sử dụng (user) khi lập trình theo một ngôn ngữ nào đó nếu dùng tới cấu trúc mảng (đối với các cấu trúc tiền định khác của ngôn ngữ cũng vậy) thì họ chỉ phải khai báo mảng (theo quy định của ngôn ngữ đó) và làm việc với các tên mảng và biến chỉ số thôi. Những vấn đề liên quan tới cấu trúc lưu trữ của mảng cũng như việc xác định địa chỉ để truy cập tới các phần tử mảng mà ta đề cập ở trên đều do chương trình dịch thực hiện. Có thể nói : các công việc "bếp núc" này chương trình dịch đã đảm nhiệm hộ, người sử dụng không phải quan tâm, thậm chí có khi không hề biết tới nữa. Tuy nhiên cần phải hiểu rằng : môi trường làm việc của người sử dụng chỉ là các tên : tên mảng, tên biến, tên hằng v. v..., nhưng thực chất đó là môi trờng địa chỉ do chương trình dịch điều hành.

2.3. ÁP DỤNG

2.3.1. Sắp xếp trên cấu trúc mảng

1. Đặt bài toán

Sắp xếp là một quá trình bố trí lại các phần tử của một tập đối tượng nào đó theo một thứ tự ấn định.

Bài toán sắp xếp xuất hiện ở rất nhiều ứng dụng, dưới nhiều dạng khác nhau :

Ở đây, ta chỉ thu hẹp lại và phát biểu dưới dạng đơn giản như sau :

Cho một dãy số A gồm n số khác nhau, mà ta coi như một vectơ với n phần tử :

$$A[1] ; A[2] ; ... ; A[n].$$

trong đó : $A[i] \neq A[j]$ nếu $i \neq j$; với $1 \leq i,j \leq n$.

Hãy sắp xếp lại các phần tử của A để chuyển nó thành một dãy số có thứ tự tăng dần (thứ tự giảm dần, cũng tương tự)

Có khá nhiều phương pháp sắp xếp khác nhau. Trước hết ta hãy xét tới một số phương pháp sắp xếp cơ bản.

2. Ba phương pháp sắp xếp cơ bản

a) Sắp xếp kiểu lựa chọn (selection–sort)

Phương pháp này ta đã xét tới ở chương I trong mục 1.3. Nó dựa vào phép lựa chọn : "Chọn số nhỏ nhất trong dãy chưa được sắp và đổi chỗ với số đang chiếm vị trí "đầu tiên" của dãy này".

Lúc đầu, dãy chưa được sắp chính là dãy A, gồm n số. Sau mỗi phép đổi chỗ dãy con chưa được sắp sẽ bớt đi một phần tử (dĩ nhiên dãy con đã được sắp sẽ tăng lên 1 phần tử).

Ta chỉ nhắc lại bằng một ví dụ.

Giả sử A bao gồm các phần tử :

32, 51, 27, 83, 66, 11, 45, 75.

Quá trình sắp xếp được minh họa qua bảng sau :

Lượt	Số nhỏ nhất	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]	A[8]
		32	51	27	83	66	(11)	45	75
1	11	11	51	(27)	83	66	32	45	75
2	27	11	27	51	83	66	(32)	45	75
3	32	11	27	32	83	66	51	(45)	75
4	45	11	27	32	45	66	(51)	83	75
5	51	11	27	32	45	51	(66)	83	75
6	66	11	27	32	45	51	66	83	(75)
7	75	11	27	32	45	51	66	75	83
Dấu		Chỉ dãy con chưa được sắp							

b) Sắp xếp kiểu thêm dần (hoặc kiểu chèn – insertion sort)

Sắp xếp kiểu thêm dần được thực hiện theo cách tương tự như cách xếp bài trên tay của người chơi bài. Khi đã có (k–1) quân bài được sắp xếp theo "đúng thứ tự" đang nằm trên tay, nếu người chơi lấy thêm quân bài thứ k thì họ sẽ phải so sánh lần lượt với một số quân bài trên tay để tìm chỗ chèn quân mới vào và sau đó trên tay họ đã có k quân bài đã được sắp xếp.

Với dãy số A thì thoạt đầu A[1] coi như một dãy con đã được sắp xếp, A[2] sẽ được xét tới : nếu A[2] < A[1], nó sẽ được chèn vào trước A[1], còn nếu A[2] > A[1], nó sẽ được giữ nguyên tại chỗ (coi như nó được chèn vào sau A[1]).

Dãy con A[1], A[2] bây giờ coi như đã được sắp xếp và A[3] được xét tới. Tuỳ theo kết quả của việc so sánh A[3] với A[2], A[1] nó lại được chèn vào hoặc sau A[2], hoặc giữa A[1] và A[2], hoặc trước A[1].

Quá trình cứ tiếp tục với A[4], A[5] ... cho tới khi toàn bộ dãy A đã được sắp xếp.

Với ví dụ về dãy số A đã nêu ở trên, quá trình sắp xếp có thể minh hoạ như sau :

lượt	số được xét	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]	A[8]
		32	(51)	27	83	66	11	45	75
1	51	32	51	(27)	83	66	11	45	75
2	27	27	32	51	(83)	66	11	45	75
3	83	27	32	51	83	(66)	11	45	75
4	66	27	32	51	66	83	(11)	45	75
5	11	11	27	32	51	66	83	(45)	75
6	45	11	27	32	45	51	66	83	(75)
7	75	11	27	32	45	51	66	75	83
Dấu		———				Chỉ dãy con đã được sắp			

Qua ví dụ trên ta thấy :

• Trừ việc "chèn" số đang được xét vào sau dãy con đã được sắp, mà thực chất là việc giữ nguyên số đó tại vị trí cũ, còn thì chèn vào giữa dãy hoặc vào trước dãy cũng đều được thực hiện trong máy bởi phép dịch chuyển một số phần tử của A về vị trí sau để lấy chỗ cho số đó.

• Phép "chèn vào trước" A[1] có thể được coi như phép "chèn vào giữa" nếu ta đưa thêm vào A một phần tử giả A[0] mà giá trị của nó nhỏ hơn bất cứ số nào của A, ta kí hiệu A[0] = $-\infty$

Sau đây là giải thuật sắp xếp kiểu thêm dần :

Procedure INSERT–SORT (A,n) ;

{Trong thủ tục này người ta dùng X làm một ô nhớ phụ để chứa khóa mới đang được xét}

1. {Khởi tạo số giả}

A[0] : = $-\infty$; {$-\infty$ ở đây là kí hiệu chỉ số nhỏ hơn mọi phần tử của A}

2. **for** i : = 2 **to** n **do begin**

X : = A[i] ; j : = i –1 ;

3. {Xác định "chỗ" cho số mới đang được xét và dịch chuyển các số cần thiết}

while X < A[j] **do begin**

A[j + 1] : = A[j] ;

j : = j – 1

end

4. {Đưa X vào đúng chỗ của nó}

A[j + 1] : = X

end ;

5. **return**

c) Sắp xếp kiểu đổi chỗ (exchange sort)

Để tiện cho việc hình dung ý chủ đạo của phương pháp, ta tưởng tượng vectơ A được đặt theo chiều thẳng đứng. Như vậy nó sẽ được duyệt từ "đáy" lên. Dọc đường nếu gặp 2 phần tử ngược thứ tự (nghĩa là A[i+1] < A[i]) thì đổi chỗ chúng cho nhau. Như vậy : sau lượt đầu, phần tử nhỏ nhất của A sẽ được chuyển lên "đỉnh". Lượt tiếp theo phần tử nhỏ nhất trong (n–1) phần tử còn lại sẽ được chuyển lên vị trí thứ hai v. v. ... Cứ tiếp tục như vậy thì sau (n–1) lượt, dãy A sẽ được sắp xếp xong.

Hình ảnh các phần tử nhỏ dịch chuyển dần lên phía trên giống như các bọt nước nổi lên trong nồi nước đun sôi ! Vì thế phương pháp này được gọi với cái tên khá đặc biệt là : *sắp xếp kiểu nổi bọt* (bubble–sort).

Ta cũng thấy ngay rằng, ở đây phép đổi chỗ đã được sử dụng như một phép chủ công. Sau đây là ví dụ minh hoạ :

	Lượt 1		2	3	4	5	6	7
A[1]	32	→ 11	11	11	11	11	11	11
A[2]	51	32	→ 27	27	27	27	27	27
A[3]	27	51	32	32	32	32	32	32
A[4]	83	27	51	→ 45	45	45	45	45
A[5]	66	83	45	51	51	51	51	51
A[6]	11	66	83	66	66	66	66	66
A[7]	45	45	66	83	→ 75	75	75	75
A[8]	75	75	75	75	83	83	83	82

Giải thuật thể hiện cách sắp xếp kiểu nổi bọt có thể viết như sau :

Procedure BUBBLE–SORT (A,n)

1. **for** i : = 1 **to** n – 1 **do**
2. {duyệt "từ đáy lên"}
   for j : = n **down to** i + 1 **do**
3. {Nếu ngược thứ tự thì đổi chỗ}
   if A[j] < A[j – 1] **then**
4. {đổi chỗ} **begin**
   X : = A[j] ;
   A[j] : = A[j – 1] ;
   A[j – 1] : = X {để cho gọn có thể kí hiệu :
   A[j] ↔ A[j – 1]}
   end
5. **return**

Nếu để ý ta sẽ thấy : giải thuật này đã phản ảnh được ý chủ đạo của phương pháp, nhưng nó còn "cứng nhắc" ở chỗ :

– Sau mỗi lượt duyệt, nó chỉ bớt đi 1 phần tử cho lượt duyệt tiếp theo, mặc dù có thể bớt đi được nhiều hơn. Như ở ví dụ trên : sau lượt thứ 3 có thể bớt đi được tới ba phần tử chứ không phải là một.

– Nó cứ thực hiện đủ (n – 1) lượt, dù có thể kết thúc sớm hơn. Như ở ví dụ trên : sau lượt thứ 4 thì dãy đã được sắp xếp xong rồi.

Vì vậy đã có những giải thuật BUBBLE–SORT khác, cải tiến hơn, nhưng ta không đề cập tới ở đây.

d) Nhận xét, đánh giá

Bây giờ ta hãy xét tới độ phức tạp về thời gian của 3 giải thuật sắp xếp đã nêu.

Trước hết cần thấy rằng : các phép toán đáng chú ý khi đánh giá thời gian thực hiện sắp xếp là phép so sánh giá trị các phần tử và phép đổi chỗ. Đổi chỗ có khi không phải thực hiện, nhưng so sánh giá trị thì bao giờ cũng phải làm (do đó các phép sắp xếp đã nêu thuộc loại sắp xếp dựa vào phép so sánh giá trị các phần tử). Vì vậy người ta coi phép so sánh các giá trị phần tử là "phép tích cực" và lấy nó làm căn cứ để đánh giá thời gian thực hiện giải thuật.

Với SELECTION–SORT (xem 1.3) ta thấy ở lượt thứ i để tìm số nhỏ nhất bao giờ cũng cần tới $C_i = (n-i)$ phép so sánh. Số lượng phép so sánh này không hề phụ thuộc gì vào tình trạng ban đầu của dãy số A cả.

Từ đó suy ra tổng các phép so sánh :

$$C_{min} = C_{max} = C_{tb} = \sum_{i=1}^{n-1}(n-i) = \frac{n(n-1)}{2}$$

Do đó : $T_t(n) = T_x(n) = T_{tb}(n) = O(n^2)$.

Với BUBBLE_SORT thì cũng tương tự như vậy.

Ta cũng có :

$$T_t(n) = T_x(n) = T_{tb}(n) = O(n^2)$$

Riêng với INSERTION–SORT thì có hơi khác.

Số lượng phép so sánh có phụ thuộc vào tình trạng ban đầu của A.

Nếu A đã được sắp xếp rồi thì khi xét tới A[k], rõ ràng A[k] > A[k–1] (nghĩa là nó cũng lớn hơn mọi số đang đứng trước nó) vì vậy chỉ cần một phép so sánh thôi.

Do đó tổng số các phép so sánh là (n–1) :

$$C_{min} = (n-1) \text{ và } T_t(n) = O(n)$$

Nhưng nếu dãy A lại có thứ tự ngược với thứ tự sắp xếp (như quy ước của ta thì đây là thứ tự giảm dần) thì ở lượt thứ i cần tới $C_i = (i-1)$ phép so sánh.

Vậy :

$$C_{max} = \sum_{i=2}^{n}(i-1) = \frac{n(n-1)}{2}$$

và $T_x(n) = O(n^2)$

Người ta cũng chứng minh được

$$T_{tb}(n) = O(n^2)$$

Tóm lại cả 3 giải thuật nêu trên đều có độ phức tạp trung bình về thời gian là $O(n^2)$. Điều đó cũng có nghĩa là tính hiệu quả của chúng sẽ giảm sút khi n lớn !

3. Sắp xếp NHANH (Quick–sort) hay sắp xếp kiểu phân đoạn (partition–sort)

Trong phần này ta sẽ xét tới một phương pháp sắp xếp khác hẳn với các phương pháp đã nêu. Nó đặc biệt không phải chỉ ở chỗ : đây là một phương pháp có hiệu lực cao về thời gian thực hiện trung bình – Như chính tên gọi nó đã bộc lộ ra – mà còn bởi nó được thực hiện bởi chiến thuật "chia để trị" và, một cách tự nhiên, điều đó đã dẫn tới một giải thuật sắp xếp đệ quy.

Không giống như phương pháp sắp xếp kiểu lựa chọn là chọn số nhỏ nhất (hoặc lớn nhất) rồi định vị cho nó, ở đây phần tử được chọn là một phần tử bất kì, mà ta gọi là "chốt" (pivot).

Khi một phần tử của mảng A đã được chọn làm "chốt" thì mọi phần tử nhỏ hơn chốt sẽ được đẩy về phía trước chốt, mọi phần tử lớn hơn chốt sẽ được đẩy về phía sau chốt. Cuối cùng các phần tử nhỏ hơn chốt sẽ tạo thành một mảng con thứ nhất, các phần tử lớn hơn chốt sẽ tạo thành mảng con thứ hai. Vị trí nằm giữa hai mảng con này chính là vị trí của chốt trong sắp xếp.

Như vậy mảng A đã được phân làm hai mảng con (hay hai phân đoạn). Đối với từng mảng con này một chiến thuật tương tự sẽ được áp dụng, và cứ như vậy cho đến khi mảng con chỉ còn là một phần tử.

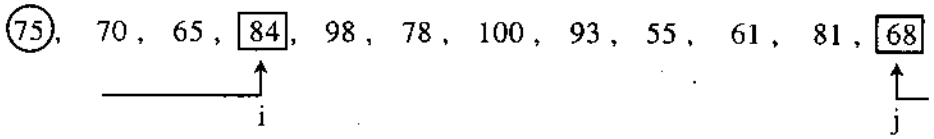
Để minh hoạ ta, xét mảng A với các phần tử như sau :

75, 70, 65, 84, 98, 78, 100, 93, 55, 61, 81, 68

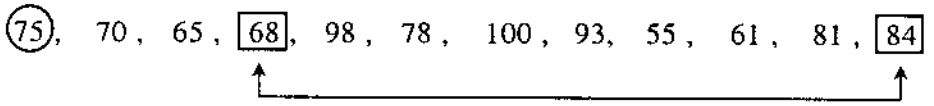
Giả sử ta chọn phần tử đầu tiên là 75 để làm "chốt". Ta phải làm sao để các số nhỏ hơn 75, cụ thể là 70, 65, 55, 61 và 68 phải chuyển về phía trước (hay phía bên trái) của 75 (nhưng không nhất thiết phải theo đúng thứ tự như trên đây) và các số lớn hơn 75 như 84, 98, 100, 93, 81 phải chuyển về phía sau

(hay phía phải) của 75. Chú ý rằng thứ tự của các phần tử thuộc các mảng con này chưa cần quan tâm đến.

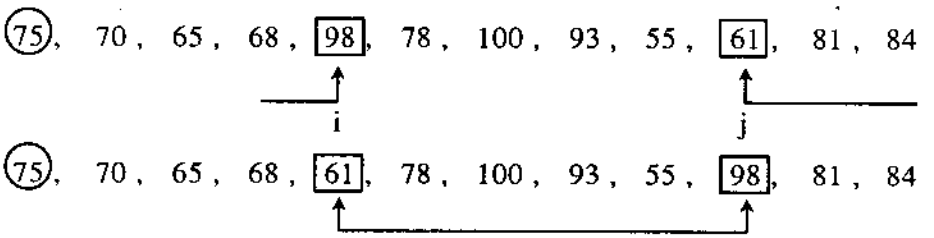
Để thực hiện việc tách mảng A thành 2 mảng con như đã nêu, ta thực hiện hai phép tìm kiếm ngược chiều nhau : một phép tìm kiếm bắt đầu từ phần tử thứ hai bên trái (vì phần tử thứ nhất ở đây ta đã chọn làm chốt rồi) để truy ra phần tử lớn hơn 75, và một phép tìm kiếm ngược lại bắt đầu từ phần tử ngoài cùng bên phải để truy ra phần tử nhỏ hơn 75. Chỉ số của phần tử truy tìm từ trái được ghi nhận bởi biến chỉ số i, như ở đây i = 4 ứng với phần tử 84. Còn chỉ số phần tử truy tìm từ phải được ghi nhận bởi biến chỉ số j, ở đây j = 12 ứng với phần tử 68. Ta thấy i < j vì vậy hai phần tử 84 và 68 được đổi chỗ cho nhau. Có thể minh hoạ :



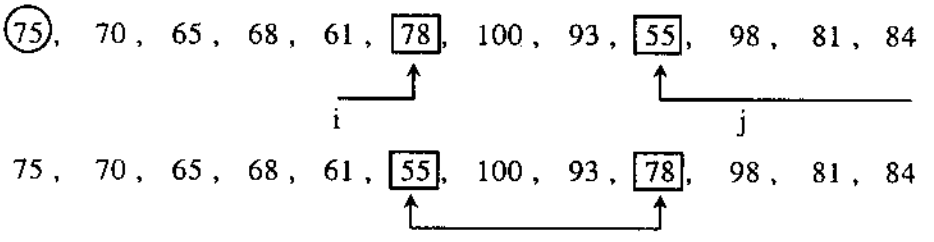
Sau khi hoán vị :



Lại tiếp tục quá trình trên, từ bên trái thì i sẽ ghi nhận : i = 5 ứng với phần tử 98, và từ bên phải thì j ghi nhận : j = 10 ứng với phần tử 61. Lúc này i < j, hai phần tử trên lại được đổi chỗ cho nhau.

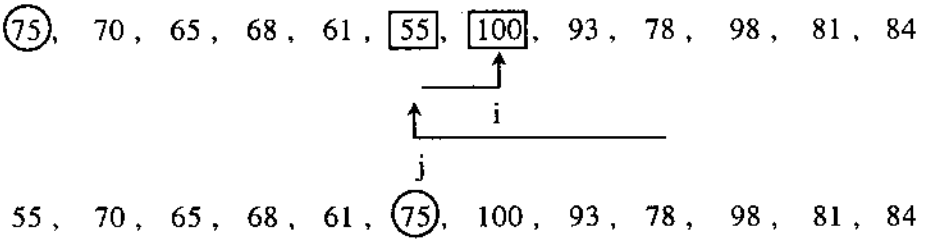


Tiếp tục, ta lại có i = 6 ứng với phần tử 78, còn j = 9 ứng với 55 và i < j nên chúng lại được đổi chỗ cho nhau



Lại tiếp tục từ bên trái, ta có i = 7 ứng với phần tử 100, còn từ bên phải thì cuối cùng j = 6, lại ứng với phần tử 55.

Lúc này i > j, việc hoán vị vị trí được thực hiện giữa 55 và chốt 75



Chú ý rằng sau lần đổi chỗ này tất cả các phần tử nhỏ hơn 75 đều nằm ở bên trái nó, còn các phần tử lớn hơn 75 đều đã nằm ở bên phải nó và như vậy A đã được chia làm hai mảng con. Mảng con thứ nhất gồm :

55, 70, 65, 68, 61

và mảng con thứ hai gồm :

100, 93, 78, 98, 81, 84

còn 75 (chốt) thì đã được đặt vào đúng vị trí của nó trong sắp xếp.

Quá trình sắp xếp lại được tiếp tục với từng mảng con, bằng một kĩ thuật tương tự.

Mỗi lượt phân đoạn như trên, được thể hiện bằng giải thuật sau :

Procedure PART (A, LB, UB, k) ;

{Ở đây A chỉ mảng một chiều cần sắp xếp, với LB là chỉ số của phần tử đầu (cận dưới) và UB là chỉ số của phần tử cuối (cận trên).

Để giải thuật thực hiện được thuận lợi, người ta còn đưa thêm vào A một phần tử A[n + 1] mà giá trị của nó lớn hơn mọi giá trị của các phần tử của A.

Giá trị ghi nhận bởi biến k, ở đây chính là chỉ số của phần tử chốt, sau mỗi lần thực hiện phân đoạn}.

1. i : = LB + 1 ; j : = UB ;

2. **while** i ≤ j **do begin**

while A[i] < A[LB] **do** i : = i +1 ;

while A[j] > A[LB] **do** j : = j −1 ;

{A[LB] ở đây được chọn làm "chốt"}

```
            if i < j then begin
                        A[i] ↔ A[j] ;
                        i.: = i +1 ;
                        j : = j +1 ;
                        end ;
                end.
```

3. k : = j ; A[LB] ↔ A[j] ; {dấu ↔ chỉ phép đổi chỗ thay cho 3 phép gán như đã nêu ở các giải thuật trước}

4. **return**.

Sau đây là giải thuật đệ quy thể hiện phương pháp sắp xếp "NHANH".

Procedure QUICK–SORT (A, LB, UB)

1. **if** LB < UB **then begin**

2. **call** PART (A, LB, UB, k) ;

3. **call** QUICK–SORT (A, LB, k–1) ;

4. **call** QUICK–SORT (A, k+1, UB)

end ;

5. **return**.

Việc sắp xếp mảng A gồm các phần tử

A[1], A[2], ..., A[n].

sẽ được thực hiện bằng lời gọi :

QUICK–SORT (A, 1, n) ;

Chú ý : 1) Với cách chọn "chốt" là phần tử đầu tiên như trong giải thuật đã nêu thì trường hợp xấu nhất xảy ra khi dãy số A vốn đã được sắp xếp theo thứ tự tăng dần (hoặc giảm dần). Lúc đó sau mỗi lượt phân đoạn ta không tách A được thành hai mảng con mà chỉ có một thôi. Điều đó có nghĩa là QUICK–SORT không hơn gì các phương pháp sắp xếp cơ bản đã nêu. Như vậy đối với QUICK–SORT $T_x(n) = O(n^2)$.

Tuy nhiên, người ta cũng chứng minh được rằng : đối với phương pháp này $T_{tb}(n) = O(n\log_2 n)$ và như vậy thì khi n lớn, hiệu lực của QUICK–SORT sẽ cao hơn hẳn các phương pháp sắp xếp cơ bản đã nêu.

2) Việc chọn "chốt" như thế nào để nâng cao hiệu quả của phương pháp, cũng là một vấn đề được quan tâm. Tuy nhiên ở đây ta không tìm hiểu sâu thêm nữa.

2.3.2. Tìm kiếm trên cấu trúc mảng

1. Đặt bài toán

Tìm kiếm một phần tử nào đó của một tập đối tượng theo một tiêu chí đã đề ra là một bài toán rất phổ biến trong tin học.

Ở đây ta xét tới một dạng đơn giản của nó :

"Cho một vectơ A bao gồm n phần tử, có giá trị là các số khác nhau :

A[1] ; A[2] ; ... ; A[n]

Cho một số X, hãy tìm xem có phần tử nào của A mà giá trị của nó bằng X hay không".

Tìm kiếm sẽ "được thỏa" khi có, hoặc "không thỏa" khi không có phần tử nào.

2. Tìm kiếm tuần tự (sequential search)

Đây là phép tìm kiếm đơn giản nhất mà ta đã nói tới trong mục 1.1. Có thể nhắc lại qua giải thuật sau :

Function SEQUENTIAL (A, n, X) ;

{Thủ tục này sẽ tìm xem trong các phần tử của A có phần tử nào có giá trị bằng X hay không. Nếu có thì nó ghi nhận lại chỉ số của phần tử đó, nếu không có thì nó ghi nhận số 0.

Trong thủ tục này người ta đưa vào một phần tử giả A[n+1] mà giá trị của nó chính là X}

1. {Khởi tạo biến chỉ số và phần tử "giả"}

   i : = 1 ;

   A[n+1] : = X ;

2. {Tìm kiếm}

   while A[i] ≠ X **do** := i := i + 1 ;

3. {Thấy hay không}

   if i : = n + 1 **then begin**

   SEQUENTIAL : = 0

   return {không thấy}

   end

```
else begin
        SEQUENTIAL : = i ;
        return {đã thấy}
        end
```

Chú ý. Đối với thủ tục hàm, tương tự như trên, bao giờ cũng có chỉ thị gán để gán giá trị cho tên hàm, trước khi kết thúc. Để tiện người ta thay cặp : chỉ thị gán giá trị cho tên hàm và **return** bằng một câu lệnh **return** (<giá trị cần gán>).

Như thủ tục trên, có thể viết

```
if i = (n+1) then return (0)
             else return (i) ;
```

3. Tìm kiếm nhị phân (binary–search)

Nếu các phần tử của A đã được sắp xếp chẳng hạn theo thứ tự tăng dần, thì phép tìm kiếm nhị phân thường được sử dụng. Thay vì so sánh X lần lượt với các giá trị của các phần tử của A như trong tìm kiếm tuần tự, người ta so sánh X với phần tử A[k] "ở giữa mảng" A :

Nếu X < A[k], tìm kiếm sẽ được thực hiện tiếp với mảng con bao gồm các phần tử ở nửa đầu của A.

Nếu X > A[k], tìm kiếm sẽ được thực hiện tiếp với các phần tử thuộc mảng con ở nửa cuối A.

Nếu X = A[k] thì tìm kiếm đã được thỏa.

Tìm kiếm sẽ không thỏa nếu mảng con trở thành rỗng.

Giải thuật sau đây thể hiện phép tìm kiếm này :

```
Function BINARY–SEARCH (A, n, X)
```

{Trong giải thuật này các biến chỉ số *l* ; *r* và *m* được sử dụng để ghi nhận chỉ số của phần tử ở đầu, ở cuối và ở giữa mảng đang được xét. Nếu tìm kiếm được thỏa, nghĩa là X = A[k] thì k được đưa ra, nếu không thì đưa ra giá trị 0}.

```
1. {Khởi tạo biến chỉ số l, r}
        l : = 1 ; r : = n ;
2. {Xác định m và tìm kiếm}
        while l ≤ r do begin
3.          m : = (l + r) div 2 ;
```

4. **if** X < A[m] **then** r : = m−1
else

5. **if** X > A[m] **then** *l* : = m+1
else return (m)
end ;

6. **return** (0)

Ví dụ : A là một vectơ bao gồm 13 phần tử :

11, 22, 30, 33, 40, 44, 55, 60, 66, 77, 80, 88, 99

1. Nếu X = 40, thì các bước thực hiện sẽ như sau :

Thoạt đầu A được xét nên *l* = 1, r = 13

• do *l* < r nên tính m = 7

X < A[7] = 55 nên r = 6, mảng con gồm các phần tử từ A[1] đến A[6] sẽ được xét,

• *l* < r nên tính m = 3

X > A[3] = 30 nên *l* = 4, mảng còn gồm các phần tử từ A[4] đến A[6] được xét.

• *l* < r nên tính m = 5

X = A[5] = 40 nên phép tìm kiếm đã được thỏa, hàm BINARY – SEARCH cho ra giá trị 5 và giải thuật kết thúc.

2. Nếu X = 75. Các bước thực hiện có thể diễn tả trong bảng sau

1	2	3	4	5	6	7	8	9	10	11	12	13	*l*	r
(11)	22	30	33	40	44	[55]	60	66	77	80	88	(99)	1	13
							(60)	66	[77]	80	88	(99)	8	13
							[(60)]	(66)					8	9
								(66)					9	9

○ : chỉ phần tử ứng với *l* và r của mảng đang xét

□ : chỉ phần tử ứng với m

Ta thấy ở đây khi *l* = r = 9 thì giải thuật kết thúc và BINARY – SEARCH mang ra giá trị bằng 0, nghĩa là tìm kiếm không thỏa.

4. Nhật xét, đánh giá

Với giải thuật tìm kiếm người ta cũng chọn phép so sánh làm "phép toán tích cực" để đánh giá thời gian thực hiện giải thuật.

• Với giải thuật SEQUENTAL, trong câu lệnh 2, phép so sánh giá trị của A[i] với X sẽ thực hiện ít nhất là 1 lần, đây là trường hợp mà X trùng ngay với A[1] và nhiều nhất là n + 1 lần. Như vậy có thể nói với giải thuật này thì $T_t(n) = 0(1)$; còn $T_x(n) = 0(n)$. Người ta cũng chứng minh được $T_{tb}(n) = 0(n)$.

• Còn với giải thuật BINARY – SEARCH thì phép so sánh X với A[m] ở câu lệnh 4, 5 cũng thực hiện ít nhất 1 lần, nghĩa là $T_t(n) = 0(1)$. Còn nhiều nhất thì sao ?

Ta thấy rằng, sau mỗi phép so sánh, số phần tử của mảng được xét tiếp theo sẽ giảm đi chừng một nửa. Như vậy sau k lần thực hiện so sánh thì số phần tử của mảng con được xét nhiều nhất sẽ là $n/2^k$, và quá trình lặp lại sẽ kết thúc khi mảng con được xét rỗng hoặc chỉ có 1 phần tử, nghĩa là : lúc đó $n/2^k \leq 1$ hay $n \leq 2^k$

$$k \geq \log_2 n$$

do đó $k = \lceil \log_2 n \rceil$

(k là số nguyên lớn hơn và sát $\log_2 n$).

Từ đấy có thể suy ra : với giải thuật này thì $T_x(n) = 0(\log_2 n)$.

Rõ ràng rằng : giải thuật tìm kiếm nhị phân "tốt hơn" giải thuật tìm kiếm tuần tự.

Người ta cũng chứng minh được rằng với giải thuật này $T_{tb}(n) = 0(\log_2 n)$ và như vậy phép tìm kiếm nhị phân có thời gian thực hiện nhỏ hơn rất nhiều so với tìm kiếm tuần tự, khi n lớn.

Câu hỏi và bài tập

2.1. Đối với các ngôn ngữ lập trình thông dụng hiện nay, cấu trúc lưu trữ của mảng là cấu trúc nào ?

2.2. Khi lập trình bằng ngôn ngữ PASCAL, nếu ta khai báo một ma trận A theo kiểu array [1..5, 1..4] of real thì máy sẽ lưu trữ như thế nào ? Minh họa cấu trúc lưu trữ bằng hình vẽ.

2.3. Đối với mảng nhiều chiều, khi lưu trữ kế tiếp cũng chỉ có 2 cách :

– Lưu trữ theo "thứ tự ưu tiên hàng".

– Lưu trữ theo "thứ tự ưu tiên cột".

Vậy thì "thứ tự ưu tiên hàng" ở đây được thể hiện qua đặc điểm gì ? "Thứ tự ưu tiên cột", ở đây được thể hiện qua đặc điểm gì ? Minh họa cách lưu trữ đối với mảng 3 chiều B mà phần tử của nó là B[i, j, k] với $1 \le i \le 2$; $1 \le j \le 4$; $1 \le k \le 3$.

2.4. Đối với 3 phương pháp sắp xếp cơ bản thì thời gian thực hiện giải thuật tương ứng có gì giống nhau ? Có gì khác nhau. So sánh chúng với QUICK-SORT

2.5. Tại sao khi đánh giá độ phức tạp về thời gian thực hiện các giải thuật sắp xếp và tìm kiếm, người ta phải nhấn mạnh điều kiện : "khi n đủ lớn".

2.6. So với "tìm kiếm tuần tự" thì "tìm kiếm nhị phân" có ưu điểm gì ? nhược điểm gì ?

2.7. Cho mảng một chiều AA[5,.50] và BB[–5..10]

a) Hãy tính số phần tử của mỗi mảng.

b) Đối với mảng AA, biết địa chỉ gốc $L_o = 300$ và mỗi phần tử (ô nhớ) ứng với $\omega = 4$ từ máy. Hãy tính địa chỉ của AA[17], AA[35].

2.8. Giả sử 3 từ máy mới đủ lưu trữ một phần tử của ma trận CC[1..6, 1..3]. Hãy tính địa chỉ của CC[4,2] biết rằng ma trận được lưu trữ theo thứ tự ưu tiên hàng và địa chỉ gốc $L_o = 1000$.

2.9. A là một ma trận tam giác được biểu diễn như hình sau :

$$A = \begin{pmatrix} a_{11} & & & & \\ a_{21} \rightarrow a_{22} & & & & \\ a_{31} \rightarrow a_{32} \rightarrow a_{33} & & & & \\ \cdots & \cdots & \cdots & \cdots & \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ a_{n1} \rightarrow a_{n2} \rightarrow a_{n3} \rightarrow \cdots \rightarrow & \cdots & \rightarrow a_{nn} & & \end{pmatrix}$$

Rõ ràng là không cần thiết phải lưu trữ các phần tử nằm ở phía trên đường chéo chính, vì chúng đều bằng 0, mà chỉ cần lưu trữ các phần tử còn lại bằng một vectơ B theo thứ tự chỉ bởi mũi tên trên hình vẽ. Nghĩa là :

$B[1] = a_{11}$; $B[2] = a_{21}$; $B[3] = a_{22}$; $B[4] = a_{31}$...

Như vậy thì vectơ B có kích thước là bao nhiêu ? ta sẽ tiết kiệm được bao nhiêu ô nhớ ?

Giả sử : phần tử a_{ij} được lưu trữ bởi B[k]. Hãy lập công thức liên hệ giữa k và i, j.

2.10. Một đơn vị quan tâm về dân số lập một bảng thống kê số lượng người sinh trong từng năm, kể từ năm 1920 đến 1970 và lưu trữ bảng đó trong máy tính bằng một mảng một chiều N[1920..1970] với N[k] có giá trị bằng số người sinh trong năm k.

Hãy viết giải thuật thực hiện :

a) In ra những năm mà không có người nào được sinh ra.

b) Tính số lượng những năm mà số người sinh ra không quá 10.

c) Tính số người đã trên 50 tuổi tính đến năm 1985.

2.11. Cho một dãy số A gồm 8 phần tử như sau :

77, 33, 44, 11, 88, 22, 66, 55

Hãy áp dụng 3 phương pháp sắp xếp cơ bản để sắp xếp A thành một dãy số tăng dần. Minh họa từng lượt thực hiện sắp xếp trong một bảng với cấu trúc như sau :

Lượt	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]	A[8]

trong đó ghi rõ giá trị của các phần tử của A ứng với từng lượt sắp xếp.

2.12. Với giải thuật BUBBLE – SORT như đã nêu, phép duyệt qua các phần tử của A được thực hiện theo kiểu "từ đáy lên" (bottom up). Người ta cũng có thể duyệt theo kiểu từ đỉnh xuống (top down)

a) Nếu vậy thì khi nào phải thực hiện đổi chỗ ? Với dãy số A như đã nêu trong bảng 2.11 thì tình trạng của A qua các lượt sẽ thay đổi thế nào ?, hãy minh họa bằng một bảng tương tự như trên.

b) Hãy viết giải thuật thực hiện sắp xếp theo cách này.

2.13. Cho một bảng danh sách các tên sinh viên đạt điểm cao trong một kì thi. Các tên này đã được sắp xếp theo thứ tự từ điển như sau :

1	AN
2	CÔNG
3	DŨNG
4	EM
5	GIAO
6	HÙNG
7	KIÊN
8	KHANG
9	LONG
10	MINH
11	PHONG
12	SƠN
13	THẮNG
14	VIỆT

Nếu áp dụng giải thuật BINARY – SEARCH để tìm kiếm tên GIAO trong danh sách thì phải thực hiện mấy lượt. Nêu rõ giá trị của biến l, r, m ứng với từng lượt.

Chương 3. DANH SÁCH (LIST)

3.1. ĐỊNH NGHĨA

Có thể nói : Trong công việc hàng ngày, danh sách là loại rất phổ dụng : danh sách những người đăng kí mua vé máy bay, danh sách những người đang chờ khám bệnh, danh sách các cuộc triển lãm sẽ được tổ chức vào năm 2004 tại Hà Nội v.v..

Tất cả chúng đều có một điểm chung là : chúng bao gồm một số hữu hạn phần tử, có thứ tự, và số lượng phần tử có thể biến động.

Có thể hình dung : danh sách A là một dãy các phần tử :

$$(a_1, a_2, ..., a_n)$$

với n là một biến.

Vectơ chính là hình ảnh của một danh sách tại một thời điểm nào đó.

Trong một danh sách luôn có phần tử đầu (phần tử thứ nhất), phần tử cuối (phần tử thứ n). Với mỗi phần tử, có phần tử trước nó (trừ phần tử đầu) và phần tử sau nó (trừ phần tử cuối). Đối với danh sách thì thường có phép bổ sung thêm phần tử mới, loại bỏ đi một phần tử cũ. Ngoài ra có thể còn có các phép như :

- Tìm kiếm một phần tử theo một tiêu chí xác định.
- Cập nhật một phần tử.
- Sắp xếp các phần tử theo một thứ tự ấn định.
- Ghép hai hoặc nhiều danh sách thành một danh sách lớn.
- Tách một danh sách thành nhiều danh sách con v.v...

3.2. LƯU TRỮ KẾ TIẾP ĐỐI VỚI DANH SÁCH

Cũng như đối với mảng, danh sách có thể được lưu trữ trong bộ nhớ bởi một vectơ lưu trữ V gồm n ô nhớ kế tiếp. Mỗi phần tử a_i của danh sách A sẽ được lưu trữ trong một ô nhớ V[i] (phần tử thứ i của V) với $1 \le i \le n$.

Nhưng do số phần tử của A thường biến động, nghĩa là kích thước n thường thay đổi, nên việc lưu trữ chỉ có thể đảm bảo được nếu biết được max(n) (giá trị lớn nhất của n). Nhưng điều này không phải lúc nào cũng xác định được mà thường chỉ là con số dự đoán. Vì vậy nếu dự trữ max(n) quá lớn thì khả năng lãng phí bộ nhớ càng nhiều vì có hiện tượng "giữ chỗ để đấy" mà chưa chắc đã dùng hết. Còn nếu max(n) lại chưa đủ so với thực tế, thì sẽ không còn chỗ để tiếp tục hoạt động. Hơn nữa, ngay cả khi đã dự trữ đủ chỗ rồi thì việc bổ sung hay loại bỏ phần tử của danh sách, mà không phải là phần tử cuối sẽ đòi hỏi phải dịch chuyển một số phần tử lùi xuống (để lấy chỗ bổ sung phần tử mới vào) hoặc tiến lên (để lấp chỗ của phần tử vừa bị loại) và điều này sẽ gây tốn phí thời gian không ít, nếu các phép toán này được thực hiện thường xuyên.

Tuy nhiên, với cách lưu trữ này, như ta đã thấy ở chương 2, ưu điểm về tốc độ truy cập lại rất rõ.

3.3. LƯU TRỮ MÓC NỐI ĐỐI VỚI DANH SÁCH

3.3.1. Giới thiệu phương pháp

Trong cách tổ chức này, mỗi phần tử của danh sách được lưu trữ trong một ô nhớ mà ta gọi là "nút" (node). Mỗi nút sẽ bao gồm một số từ máy kế tiếp, đủ để lưu trữ các thông tin cần thiết, đó là : thông tin ứng với mỗi phần tử của danh sách và địa chỉ của nút tiếp theo. Như vậy quy cách của mỗi nút có thể hình dung như sau :

INFO	LINK

nghĩa là : mỗi nút gồm có 2 trường.

Trường INFO chứa thông tin ứng với phần tử của danh sách.

Trường LINK chứa địa chỉ của nút tiếp theo (nút sau nó).

Riêng nút cuối cùng thì không có nút tiếp theo nữa nên trường LINK của nó phải chứa một "địa chỉ đặc biệt", chỉ mang tính chất quy ước, dùng để đánh dấu nút kết thúc danh sách chứ không như các địa chỉ ở các nút khác, ta gọi nó là "địa chỉ null" hay "mối nối không".

Tất nhiên, để có thể truy cập được vào mọi nút trong danh sách thì phải biết được địa chỉ của nút đầu tiên, hay nói một cách khác là phải "nắm được" con trỏ L, trỏ tới nút đầu tiên này.

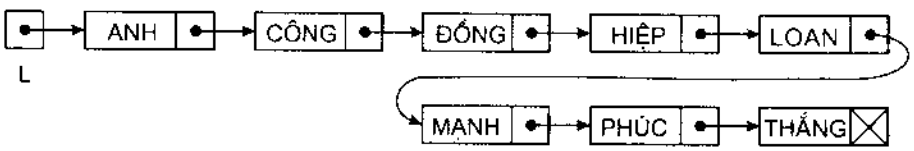
Ví dụ như ta có một danh sách tên các sinh viên vừa đạt điểm 10 trong kì thi môn "cấu trúc dữ liệu và giải thuật", nay ta muốn công bố các tên đó theo thứ tự "từ điển", ta có thể tổ chức theo kiểu móc nối như sau :

8

STT	INFO	LINK
1	MẠNH	7
2	HIỆP	4
3	THẮNG	0
4	LOAN	1
5	CÔNG	6
6	ĐỒNG	2
7	PHÚC	3
8	ANH	5

Ở đây "mối nối" đã được thay bằng số thứ tự (có thể coi đây là địa chỉ tương đối), "mối nối không" đã được kí hiệu bằng số 0 (không có số thứ tự nào bằng 0 cả). Địa chỉ L ở đây là 8, ứng với nút đầu tiên của danh sách.

Có thể minh họa danh sách móc nối này bằng hình ảnh như sau



Mũi tên → : chỉ "mối nối" : địa chỉ nút tiếp theo

Dấu × : chỉ "mối nối không" (địa chỉ null)

HÌNH 3.1.

Nút đầu tiên của danh sách này có địa chỉ là 8, dựa vào trường LINK ta biết tiếp theo là nút có địa chỉ 5, sau nút 5 là nút 6, sau nút 6 là nút 2, sau nút 2 là nút 4, sau nút 4 là nút 1, sau nút 1 là nút 7, sau nút 7 là nút 3, sau nút 3 không còn nút nào : 3 là nút kết thúc danh sách.

Cần chú ý là : một cách tổng quát thì mỗi nút của danh sách móc nối có thể nằm ở bất kì chỗ nào trong bộ nhớ, và như vậy thì địa chỉ nằm ở trường LINK của mỗi nút là địa chỉ thực của nút tiếp theo (ví dụ trên chỉ là

một minh họa đơn giản). Người ta cũng quy ước : danh sách *rỗng* là danh sách không có chứa nút nào. Lúc đó L = null.

Nếu p là một con trỏ, trỏ đến một nút bất kì trong danh sách móc nối thì phần thông tin của phần tử tương ứng sẽ được kí hiệu là INFO (p) ; phần địa chỉ nút tiếp theo sẽ được kí hiệu là LINK(p).

Tới đây, còn một vấn đề đặt ra nữa là : Làm sao để có thể nhận được một nút để sử dụng khi vận hành trên danh sách móc nối, chẳng hạn như khi cần bổ sung thêm một nút mới vào danh sách, hay khi tạo nên danh sách mới v.v.. hoặc khi một nút bị loại bỏ đi thì trả nó về đâu.

Tất nhiên phải có một vùng lưu trữ các nút chưa dùng tới, mà ta sẽ gọi là "danh sách chỗ trống (list of available space) và phải có một cơ chế để phân vùng nhớ cho phép này thành các nút với các trường dữ liệu như đã nêu, cũng như để "cấp phát" các nút trống khi có yêu cầu và "thu hồi" chúng lại khi chúng bị "thải ra". Ở đây ta sẽ không đi sâu vào việc tạo dựng nên "danh sách chỗ trống", với các nút có quy cách ấn định cũng như việc thực hiện "cấp phát" và "thu hồi" chỗ trống như thế nào. Ta coi như các chương trình thể hiện các cơ cấu và cơ chế nói ở trên đã có sẵn và khi cần ta chỉ việc sử dụng thôi. Cụ thể là :

Câu lệnh **call** New(p) ; sẽ cho ta một nút trống với quy cách ấn định, có địa chỉ là p để sử dụng còn câu lệnh : **call** dispose (p) ; sẽ trả lại cho "danh sách chỗ trống" nút có địa chỉ là p.

Sau đây ta sẽ xét tới một số giải thuật thực hiện một số phép xử lí trên danh sách móc nối.

3.3.2. Một số phép toán trên danh sách móc nối

1. Duyệt qua một danh sách móc nối

Phép duyệt một danh sách móc nối là phép "thăm" từng nút trong danh sách đó, mỗi nút chỉ thăm 1 lần, và ở mỗi nút thực hiện một phép xử lí nào đấy.

Cụ thể ở đây bài toán được phát biểu như sau :

"Cho một danh sách móc nối, có con trỏ L trỏ tới nút đầu tiên trong danh sách. Hãy in lần lượt phần thông tin ở từng nút trong danh sách đó".

Ta thấy ngay là phải duyệt quạ danh sách trỏ bởi L và ứng với một nút p nào đó thì in INFO(p).

Dĩ nhiên ta phải dùng một biến p để ghi nhận địa chỉ của từng nút, trong phép duyệt, thoạt đầu p lấy giá trị của L, sau đó p lần lượt lấy giá trị là địa chỉ của các nút tiếp theo (p được gọi là *biến trỏ* : *variable pointer*)

Sau đây là giải thuật :

Procedure TRAVERS (L) ;

1. **if** L = null **then return** ; {nếu danh sách rỗng thì không làm gì} ;

2. p := L ;

3. **while** p ≠ null *do* **begin**

write (INFO(p)) ;

p := LINK (p)

end ;

4. **return**

2. Bổ sung thêm một nút vào danh sách móc nối

"Cho một danh sách móc nối có con trỏ L trỏ tới nút đầu tiên. Hãy bổ sung thêm một nút mới vào trước nút đầu tiên này (nếu có). Thông tin của nút mới này là A".

Cần chú ý là : nếu danh sách rỗng, nghĩa là L = null thì danh sách không có nút đầu tiên ; nút mới bổ sung sẽ trở thành nút duy nhất của danh sách. Trong trường hợp nào, thì sau phép bổ sung, L cũng sẽ là địa chỉ của nút mới. Do đó ta có thủ tục :

Procedure INSERT (L, A) ;

1. {Tạo lập nút mới}

call New(p) ;

INFO(p) := A ; {gán A vào trường INFO}

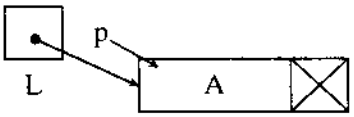
LINK(p) := L ; {gán địa chỉ L vào trường LINK}

2. {Bổ sung}

L := p ; {L bây giờ là địa chỉ nút mới bổ sung vào}

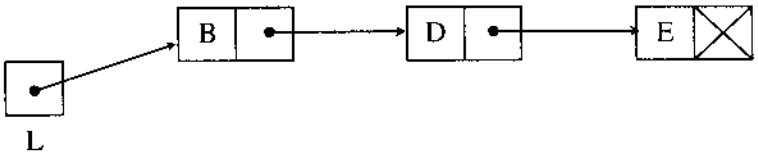
3. **return**.

Ta thấy giải thuật trên xử lí được cả trường hợp danh sách rỗng, lúc đó L = null và LINK(p) := L tức là LINK(p) : = null. Lúc đầu danh sách rỗng nhưng sau phép bổ sung danh sách có hình ảnh như sau :

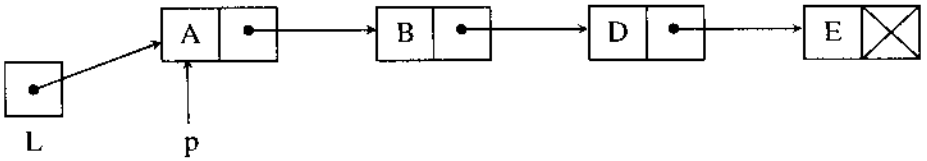


Còn trường hợp tổng quát thì danh sách có dạng :

Trước :



Sau :



(các chữ in A, B, ... tượng trưng cho phần thông tin ứng với mỗi nút)

HÌNH 3.2

3. Loại bỏ một nút ra khỏi danh sách móc nối

"Cho danh sách móc nối trỏ bởi L như trên, giả sử rằng danh sách này không rỗng. Hãy loại bỏ nút cuối cùng của danh sách".

• Rõ ràng là nếu danh sách chỉ có 1 nút thì nó cũng là nút cuối cùng và sau phép loại bỏ thì danh sách trở thành rỗng.

Còn trường hợp danh sách có từ 2 nút trở lên thì không những phải tìm đến nút cuối cùng p với đặc điểm nhận biết là LINK(p) = null, mà còn phải tìm đến nút đứng trước nút cuối cùng nữa, để sửa mối nối ở nút đó. Vì vậy trong giải thuật dưới đây ta sẽ dùng 2 biến trỏ : p và q để cuối cùng ghi nhận địa chỉ nút cuối danh sách và nút đứng trước nút cuối này.

Procedure DELETE (L) ;

1. **if** LINK(L) = null **then begin**
   call dispose(L);
   L:=null;
   return
   end;

2. {Khởi tạo các biến trỏ p và q}

p := L ; q := L ;

3. {Tìm đến nút cuối danh sách}

while LINK(p) ≠ null **do** p := LINK(p) ;

4. {Tìm đến nút đứng trước nút cuối danh sách}

while LINK(q) ≠ p **do** q := LINK(q) ;

5. {loại nút p ra khỏi danh sách, sửa nút q thành nút cuối danh sách}

call dispose(p) ;

LINK(q) := null ;

6. **return**

Chú ý. Ở thủ tục trên ta phải dùng 2 vòng lặp 3, 4, để xác định nút cuối danh sách và nút đứng trước nó.

Tuy nhiên, ta có thể viết gọn hơn bằng một câu lệnh **while** như sau :

while LINK(p) ≠ null **do begin**

q := p ;

p := LINK(p) ;

{q giữ lại địa chỉ cũ của p, trước khi cho p lấy địa chỉ nút tiếp theo}

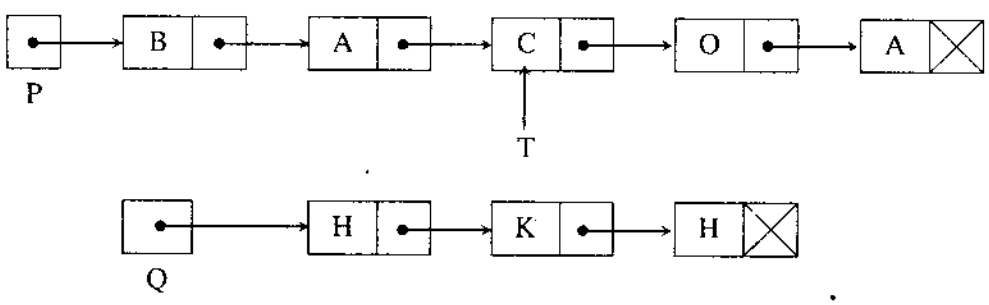
end ;

4. Ghép hai danh sách móc nối thành một

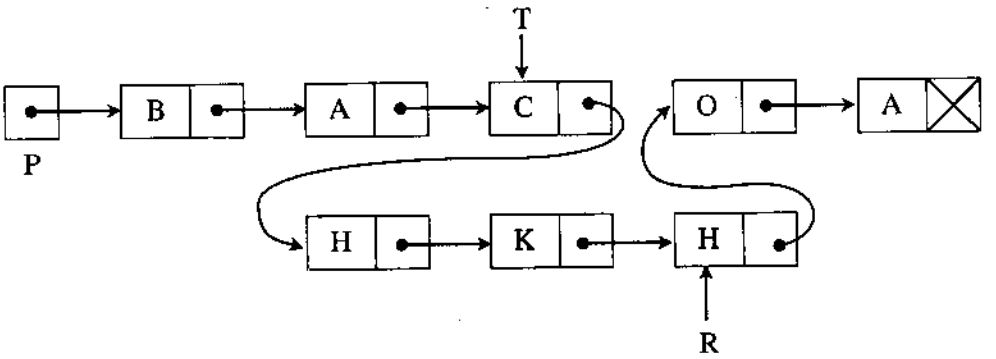
"Cho hai danh sách móc nối lần lượt được trỏ bởi P và Q. Biết rằng cả hai danh sách này đều không rỗng và trong danh sách P có một nút được trỏ bởi con trỏ T (có địa chỉ là T). Hãy viết giải thuật chèn danh sách Q vào sau nút trỏ bởi T (cuối cùng sẽ được một danh sách lớn hơn mà con trỏ trỏ tới nút đầu tiên của nó vẫn là P)"

• Có thể hình dung danh sách trước và sau phép ghép qua hình 3.3 :

Trước :



Sau :



HÌNH 3.3

Procedure IN – LIST (P, Q, P)

{P, Q là hai con trỏ input, P là con trỏ output}

1. {Tìm đến nút cuối cùng của danh sách Q}

R := Q ;

while LINK (R) ≠ null **do** R := LINK (R) ;

2. {ghép nối}

LINK(R) := LINK(T) ;

LINK(T) := Q

3. **return**.

3.3.3. Một số dạng khác của danh sách móc nối

Trong một số trường hợp, để tăng thêm tính linh hoạt cho việc xử lí trên danh sách móc nối, người ta có thay đổi đôi chút quy cách để tạo nên những danh sách móc nối cải tiến như : danh sách nối vòng, danh sách nối kép sau đây.

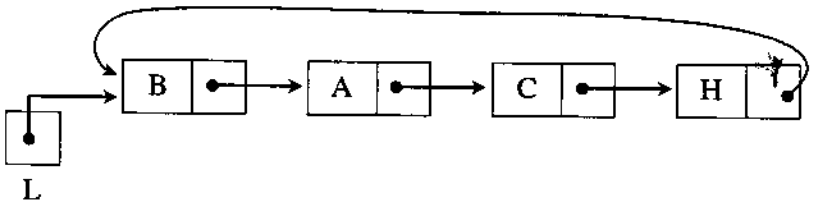
1. Danh sách nối vòng

Kiểu danh sách này chỉ khác với danh sách móc nối đã nêu, ở chỗ : mối nối ở nút cuối cùng không phải là "mối nối không" mà lại là địa chỉ của nút đầu tiên trong danh sách.

Thực ra với cách tổ chức này thì nút nào cũng có thể coi là nút đầu tiên được và chỉ cần biết địa chỉ của bất kì nút nào cũng có thể truy cập được vào mọi nút trong danh sách (ta quy ước gọi nút mà ta biết địa chỉ – hay nói một cách khác là nút có con trỏ L trỏ tới nó là nút đầu tiên).

Với danh sách móc nối đã nêu ở trên (mà từ đây, để dễ phân biệt ta sẽ gọi là "*danh sách nối thẳng*" hay "*danh sách nối đơn*") thì không thể có được điều đó. Ta chỉ có thể truy cập được vào mọi nút của danh sách nếu biết địa chỉ nút đầu tiên thôi.

Hình ảnh của danh sách nối vòng sẽ như sau :



HÌNH 3.4

2. Danh sách nối kép

Mỗi nút trong danh sách này lại có hai trường con trỏ, theo quy cách như sau :

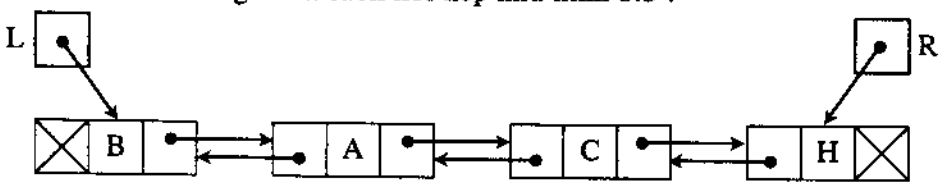
Ngoài trường INFO giống như trước đây đã nói còn có trường LPTR để ghi nhận địa chỉ của nút ở bên trái (nút trước nó) và trường RPTR để ghi nhận địa chỉ nút ở bên phải (nút sau nó).

Như vậy từ một nút không phải chỉ biết địa chỉ của nút sau nó, như trong các danh sách nối đơn, hay nối vòng, mà còn biết cả địa chỉ nút trước nó. (Rõ ràng phép loại bỏ một nút ra khỏi danh sách nối kép sẽ thực hiện dễ dàng hơn với danh sách nối đơn vì ta có thể tìm ngay được địa chỉ của nút trước nút bị loại).

Cần chú ý rằng ở đây nút đầu tiên (mà ta gọi là nút cực trái) không có nút trước nó nên LPTR có giá trị null ; đối với nút cuối cùng (nút cực phải) thì RPTR cũng có giá trị null.

Tất nhiên, để có thể truy cập vào danh sách theo cả hai chiều thì phải "nắm được" hai con trỏ : con trỏ L, trỏ tới nút cực trái và con trỏ R trỏ tới nút cực phải ? Ta cũng quy ước khi danh sách nối kép rỗng thì L = R = null.

Có thể hình dung danh sách nối kép như hình 3.5 :



HÌNH 3.5.

3.4. ÁP DỤNG : Bài toán cộng hai đa thức.

Ta xét bài toán cộng hai đa thức có dạng tổng quát như sau :

$$P(x) = a_n x^n + a_{n-1} x^{n-1} + ... + a_1 x + a_o$$

Chẳng hạn, ta có :

$$A(x) = 2x^8 - \quad 5x^7 + 3x^2 + 4x - 7$$

và $$B(x) = 6x^8 + 5x^7 - 2x^6 + x^4 - 8x^2$$

khi cộng hai đa thức này ta được đa thức tổng :

$$C(x) = 8x^8 - 2x^6 + x^4 - 5x^2 + 4x - 7.$$

Cách biểu diễn đa thức :

Để biểu diễn đa thức, trong máy tính, ta có thể chọn hoặc lưu trữ kế tiếp, hoặc lưu trữ móc nối.

a) Với cách lưu trữ kế tiếp, nghĩa là lưu trữ phần thông tin cần thiết ứng với mỗi số hạng của đa thức bởi một phần tử của vectơ lưu trữ. Chú ý rằng : mỗi số hạng của đa thức có dạng : $a_i x^i$ với $n \geq i \geq 0$, nghĩa là ta phải xác định được hệ số a_i và số mũ i. Nhưng mỗi phần tử của vectơ lưu trữ, thường chỉ ghi nhận một giá trị thôi. Vì vậy nếu một phần tử của vectơ lưu trữ chỉ ghi nhận giá trị của hệ số a_i thì số mũ i phải ẩn dụ trong thứ tự của phần tử đó. Và điều đó còn tùy thuộc vào việc ta ấn định kích thước chung cho vectơ lưu trữ như thế nào. Chẳng hạn, ta quy ước là mỗi vectơ V có kích thước bằng 9 thì ta chỉ có thể lưu trữ được đa thức với số mũ tối đa là 8 và V[1] lưu trữ giá trị của a_8

V[2] lưu trữ giá trị của a_7

.....

V[8] lưu trữ giá trị của a_o

Bất kì đa thức nào cũng phải được lưu trữ theo đúng quy ước đó.

Như với 2 đa thức A(x) và B(x) đã nêu ở trên thì các vectơ lưu trữ chung sẽ có hình ảnh như sau :

	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]	A[8]	A[9]
A :	2	–5	0	0	0	0	3	4	–7

	B[1]	B[2]	B[3]	B[4]	B[5]	B[6]	B[7]	B[8]	B[9]
B :	6	5	–2	0	1	0	–8	0	0

HÌNH 3.6

Rõ ràng là với cách tổ chức lưu trữ như thế này thì phép cộng hai đa thức chỉ là phép cộng hai vectơ thôi, như ở ví dụ trên : với A(x) và B(x) đã cho thì sau khi thực hiện phép cộng 2 vectơ A và B ta sẽ có vectơ biểu diễn đa thức tổng C(x) có dạng :

C :

8	0	−2	0	1	0	−5	4	−7
C[1]	C[2]	C[3]	C[4]	C[5]	C[6]	C[7]	C[8]	C[9]

HÌNH 3.7

Ở đây ta thấy rõ ưu điểm là : giải thuật thực hiện phép cộng đa thức quá đơn giản. Nhưng bên cạnh đó cũng xuất hiện một nhược điểm rất lớn.

Trước hết là kích thước ấn định cho vectơ lưu trữ phụ thuộc vào số mũ lớn nhất của số hạng có trong đa thức. Nếu một trong hai đa thức có mũ cao, nhưng lại ít số hạng, chẳng hạn :

$$A(x) = 15x^{1000} - x$$

thì rõ ràng phải có vectơ với kích thước bằng 1001 phần tử để thực hiện lưu trữ và như vậy thì quá lãng phí bộ nhớ. Còn nếu ta hạn chế kích thước lại thì hiệu lực của giải thuật cũng bị hạn chế theo. Như với quy ước, kích thước vectơ bằng 9 trong ví dụ trên thì giải thuật chỉ có hiệu lực với các đa thức mà số hạng có số mũ nhỏ hơn hoặc bằng 8 thôi.

Chú ý : Ở trên, ta quy ước : đa thức được biểu diễn theo số mũ giảm dần của số hạng. Nếu ta biểu diễn theo thứ tự ngược lại, tất nhiên quy ước lưu trữ sẽ phải thay đổi theo.

Để khắc phục các nhược điểm của lưu trữ kế tiếp như đã nêu trên, ta có thể dùng cách lưu trữ móc nối.

b) Với lưu trữ móc nối thì đa thức có thể biểu diễn dưới dạng danh sách nối đơn mà mỗi nút của nó có quy cách như sau :

COEF	EXP	LINK

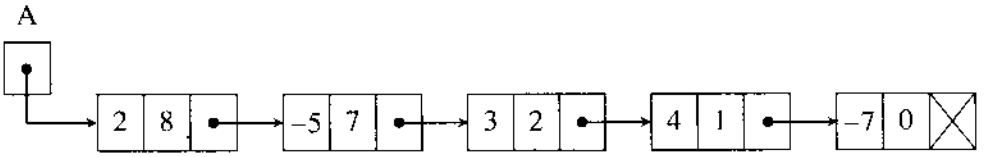
Như vậy là mỗi nút có 3 trường :

Trường COEF chứa hệ số khác không của mỗi số hạng trong đa thức.

Trường EXP chứa số mũ tương ứng.

Trường LINK chứa địa chỉ nút tiếp theo trong danh sách.

Như với đa thức A(x) ở trên thì danh sách biểu diễn nó có dạng :



(ở đây A là con trỏ, trỏ tới nút đầu tiên của danh sách)

HÌNH 3.8

Với cách lưu trữ này thì đa thức có bao nhiêu số hạng với hệ số khác không, danh sách nối đơn biểu diễn nó sẽ có bấy nhiêu nút.

Bây giờ ta xét tới giải thuật thực hiện cộng hai đa thức A(x) và B(x). Giả sử rằng danh sách biểu diễn chúng đã được tạo lập trong máy và đã được trỏ lần lượt bởi con trỏ A và B. Ta sẽ gọi C là con trỏ, trỏ tới danh sách biểu diễn đa thức tổng.

Rõ ràng phải dùng 2 biến trỏ p và q để thăm lần lượt các nút, khi duyệt qua hai danh sách. Ta thấy có những tình huống như sau :

1) Nếu EXP (p) = EXP (q) ta sẽ phải thực hiện cộng giá trị ở trường COEF của hai nút đó. Nếu giá trị tổng khác không thì phải tạo ra nút mới để biểu diễn số hạng tương ứng và bổ sung vào (gọi tắt là "gắn vào") danh sách tổng.

2) Nếu EXP (p) > EXP (q), nghĩa là trong danh sách B không có số hạng cùng số mũ với p như trong danh sách A (Ngược lại EXP(q) > EXP (p) thì xử lí cũng tương tự). Như vậy sẽ phải sao chép thông tin ở nút p vào một nút mới và gắn vào danh sách tổng.

3) Nếu một trong hai danh sách kết thúc trước thì các nút còn lại của danh sách kia sẽ được sao chép lần lượt vào nút mới và "gắn vào" danh sách tổng.

Mỗi lần một nút mới được tạo ra thì nó được gắn vào sau nút cuối cùng của danh sách tổng (ta gọi tắt là nút "đuôi" của danh sách tổng). Như vậy, phải thường xuyên nắm được địa chỉ của nút đuôi này, ta gọi địa chỉ đó là d. Ta thấy ngay rằng các công việc :

"– Xin cấp phát một nút mới

– Sao chép thông tin vào trường COEF và EXP của nút đó.

– "gắn" nút mới này vào sau nút trỏ bởi d, và lại biến nó thành nút đuôi mới"

sẽ được lặp lại nhiều lần trong quá trình xử lí. Vì vậy ta sẽ viết nó dưới dạng một chương trình con và sẽ "gọi nó" khi cần sử dụng. Sau đây là thủ tục :

Procedure ATTACH (H, M, d)

{H là nội dung sẽ được sao chép vào trường COEF, nó biểu thị hệ số của số hạng mới trong danh sách tổng. Còn M thì biểu thị số mũ}.

1. **call** New(p) ; {xin cấp phát một nút mới p}

2. {sao chép thông tin}

COEF(p) := H ;

EXP(p) := M ;

3. {gắn vào danh sách tổng và biến nó thành nút đuôi mới}

LINK(d) := p ;

d := p

4. **return**

Thủ tục : cộng hai đa thức sẽ được thể hiện như sau :

Procedure ADDPOL (A, B, C) ;

1. {Khởi tạo các biến trỏ}

p := A ; q := B ;

2. {tạo "nút đuôi giả" để ngay từ đầu đã có khả năng sử dụng thủ tục ATTACH}

call New(C) ; d := C ;

3. {xử lí hai tình huống đầu}

while p ≠ null **and** q ≠ null **do**
case
EXP (p) = EXP(q) : x := COEF(p) + COEF(q) ;
if x ≠ 0 **then**
call ATTACH (x, EXP(p), d) ;
p := LINK(p) ; q : = LINK(q) ;
EXP(p) > EXP (q) : **call** ATTACH (COEF(p), EXP(p), d) ;
p := LINK(p) ;
EXP(p) < EXP(q) : **call** ATTACH (COEF(q), EXP(q), d) ;
q := LINK(q) ;
end case ;

4. {xử lí trường hợp danh sách A kết thúc trước}

while q ≠ null **do begin**

call ATTACH (COEF(q), EXP(q), d) ;

q := LINK(q)

end ;

5. {xử lí trường hợp danh sách B kết thúc trước}

while p ≠ null **do begin**

call ATTACH (COEF(p), EXP(p), d) ;

p := LINK(p)

end ;

6. {kết thúc danh sách tổng C}

LINK(d) := null ;

7. {Loại bỏ "nút đuôi giả" và cho C trỏ tới danh sách tổng}

t := C ; C := LINK(C) ; **call** dispose (t) ;

8. **return**.

Câu hỏi và bài tập

3.1. Hãy nêu tóm tắt ưu điểm và nhược điểm của phương pháp lưu trữ kế tiếp và lưu trữ móc nối đối với danh sách.

3.2. Việc chèn một phần tử vào "giữa" danh sách (không phải là trước nút đầu và sau nút cuối) khi lưu trữ kế tiếp có gì khác so với khi lưu trữ móc nối ?

3.3. Với danh sách nối đơn, việc truy cập vào phần tử nào được thực hiện trực tiếp ?

3.4. Để xác định được một nút "đứng trước" một nút đã cho trong danh sách nối đơn thì phải biết những địa chỉ nào và phải làm thế nào ? Với danh sách nối vòng, có như vậy không ?

3.5. Cho một danh sách nối đơn, có con trỏ LIST trỏ tới nút đầu tiên của danh sách này. Hãy viết giải thuật thực hiện :

a) Cộng thêm một số A vào số đang chứa tại trường INFO của mỗi nút.

b) Đếm số lượng các nút có trong danh sách đó.

c) Đếm số lượng các nút đang chứa số dương thuộc danh sách đó (giả sử các số chứa trong mỗi nút là số đại số khác không).

3.6. Cho một danh sách nối đơn có con trỏ P trỏ tới nút đầu tiên của nó. Biết rằng danh sách này không rỗng. Hãy viết giải thuật :

a) Bổ sung một nút mới vào sau nút có địa chỉ T (hay : con trỏ T trỏ tới nó) đang có trong danh sách đó.

b) Bổ sung một nút mới vào trước nút trỏ bởi T đang có trong danh sách đó.

Biết rằng phần thông tin dành cho nút mới đang chứa trong ô có địa chỉ là X.

3.7. Cho một danh sách nối đơn có con trỏ Q trỏ tới nút đầu tiên của danh sách. Biết rằng danh sách này không rỗng. Hãy viết giải thuật :

a) Loại bỏ nút đầu tiên của danh sách.

b) Loại bỏ nút trỏ bởi T đang có trong danh sách. Biết rằng T không phải là địa chỉ nút đầu tiên và cũng không phải là địa chỉ nút cuối cùng.

3.8. Cho ba danh sách nối đơn, lần lượt có nút đầu tiên được trỏ bởi L1, L2, L3. Hãy viết giải thuật :

a) Ghép danh sách L2 vào sau danh sách L1 và cho L trỏ tới danh sách tổng hợp.

b) Ghép danh sách L3 vào trước L2 và L1 vào sau L2 và cho L trỏ tới danh sách tổng hợp.

Giả sử rằng cả ba danh sách này đều không rỗng.

3.9. Cho một danh sách nối đơn có con trỏ P trỏ tới nút đầu tiên của nó, danh sách này không rỗng. Cho con trỏ Q trỏ tới một nút đang có trong danh sách. Hãy viết giải thuật tách danh sách này thành hai danh sách con. Danh sách thứ nhất sẽ được trỏ bởi P.

Danh sách thứ hai sẽ được trỏ bởi Q (nút trỏ bởi Q sẽ là nút đầu tiên của danh sách thứ hai).

3.10. Cho một danh sách nối đơn, không rỗng, có con trỏ LIST trỏ tới nút đầu tiên. Biết rằng trường INFO của mỗi nút đều chứa một số dương. Hãy viết giải thuật :

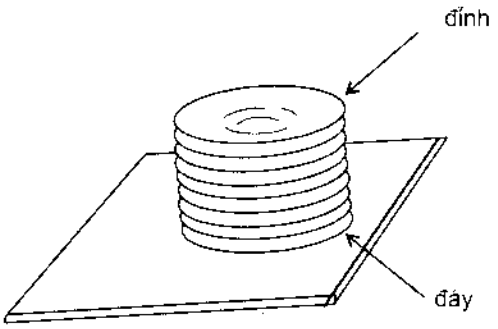
Tính giá trị trung bình của các số chứa trong danh sách.

Chương 4. NGĂN XẾP (STACK) VÀ HÀNG ĐỢI (QUEUE)

4.1. ĐỊNH NGHĨA STACK

Stack là một kiểu danh sách đặc biệt mà phép bổ sung và phép loại bỏ luôn thực hiện ở một đầu ; được gọi là đỉnh (top).

Có thể hình dung cơ cấu của stack như một chồng đĩa. Đưa thêm một đĩa mới vào thì đặt nó vào đầu phía trên (đỉnh), lấy một đĩa ra thì cũng chỉ lấy được từ đầu đó. Đĩa đưa vào sau cùng, chính là đĩa đang nằm ở đỉnh, và nó cũng chính là đĩa sẽ được lấy ra trước tiên. Còn đĩa đưa vào đầu tiên lại đang ở vị trí được gọi là đáy (bottom) và nó chính là đĩa được lấy ra sau cùng.



HÌNH 4.1

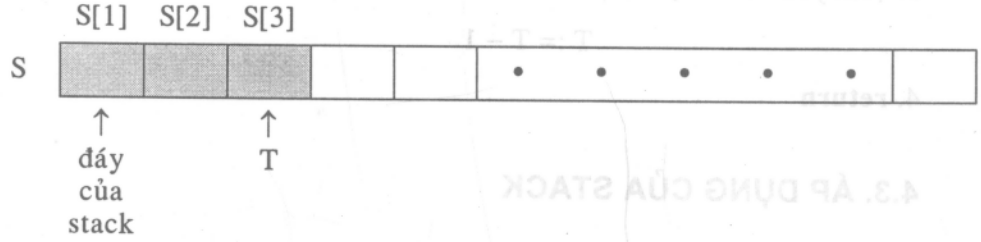
Như vậy stack hoạt động theo cơ chế : "vào – sau – ra – trước" (last – in – first – out). Do đó stack còn được gọi là danh sách kiểu LIFO. Stack có thể rỗng, nghĩa là không có phần tử nào

4.2. LƯU TRỮ KẾ TIẾP ĐỐI VỚI STACK

Đó chính là cách cài đặt stack bởi một vectơ lưu trữ S, bao gồm n ô nhớ kế tiếp nhau. Như vậy phần tử ở đỉnh stack sẽ được định vị bởi một chỉ số i nào đó, mà ta coi như một địa chỉ tương đối (địa chỉ thực – địa chỉ tuyệt đối – sẽ được xác định như đã nêu ở mục 2.2.2 thuộc chương 2).

Có thể coi i như giá trị của một con trỏ T, trỏ tới đỉnh stack. Người ta quy ước T = 0 nghĩa là stack rỗng. Như vậy T = i thì stack có i phần tử. Rõ ràng $0 \le T \le n$, khi T = n thì stack đã đầy, lúc đó nếu có phép bổ sung một phần tử mới vào stack thì sẽ không thực hiện được, vì "không còn chỗ" ; ta nói là có hiện tượng TRÀN (overflow) và tất nhiên việc xử lí phải ngừng lại. Còn nếu

T = 0, nghĩa là stack đã rỗng, mà lại có phép loại bỏ một phần tử ra khỏi stack thì phép xử lí này cũng không thực hiện được ; Ta nói : có hiện tượng CẠN (underflow). Sau đây là hình ảnh cài đặt của stack với 3 phần tử.



HÌNH 4.2

Khi bổ sung một phần tử mới vào thì T tăng lên 1, còn khi loại bỏ một phần tử ra khỏi stack thì T giảm đi 1.

Hai thủ tục dưới đây thể hiện hai phép xử lí này.

Procedure PUSH (S, T, X) ;

{Thủ tục này thực hiện việc bổ sung phần tử X vào stack, cài đặt bởi vectơ S mà T đang trỏ tới đỉnh}

1. {Kiểm tra xem stack đã đầy chưa}

if T = n **then begin**
write ('stack sẽ TRÀN') ;
return
end ;

2. {chuyển con trỏ}

T : = T + 1 ;

3. {nạp X vào}

S[T] := X ;

4. **Return**

Procedure POP (S, T, Y) ;

{Thủ tục này thực hiện việc loại phần tử đang ở đỉnh stack ra khỏi stack và bảo lưu thông tin của phần tử này vào Y)

1. {Kiểm tra xem stack có cạn không}

if T = 0 **then begin**
write ('stack dã CẠN') ;
return
end ;

2. {Bảo lưu thông tin của phần tử sẽ bị loại}

Y :=S[T] ;

3. {chuyển con trỏ}

T := T – 1

4. **return**

4.3. ÁP DỤNG CỦA STACK

4.3.1. Bài toán đổi cơ số từ thập phân sang nhị phân

Ta đã biết : dữ liệu lưu trữ trong bộ nhớ của MTDT đều được biểu diễn dưới dạng số nhị phân (với 2 chữ số 0 và 1). Như vậy nghĩa là các số thập phân xuất hiện trong chương trình đều phải chuyển đổi sang nhị phân trước khi xử lí và các kết quả dưới dạng số nhị phân sẽ lại được đổi sang thập phân để hiển thị cho người dùng biết (vì người dùng vốn đã quen với số thập phân rồi).

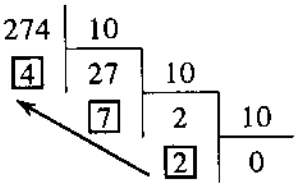
Một cách tổng quát : số thập phân bao gồm hai phần, phần nguyên và phần phân số thập phân - mà ta vẫn quen gọi là phần lẻ. Việc chuyển đổi sang số nhị phân, ứng với hai phần, có khác nhau. Ở đây ta chỉ xét tới việc chuyển phần nguyên thôi, hay nói một cách khác : ta sẽ xét tới việc chuyển đổi một số nguyên ở dạng thập phân sang dạng nhị phân.

Trước hết, cần thấy rằng

Ở dạng thập phân số 274 biểu diễn cho con số có giá trị là :

$$2.10^2 + 3.10^1 + 4.10^0.$$

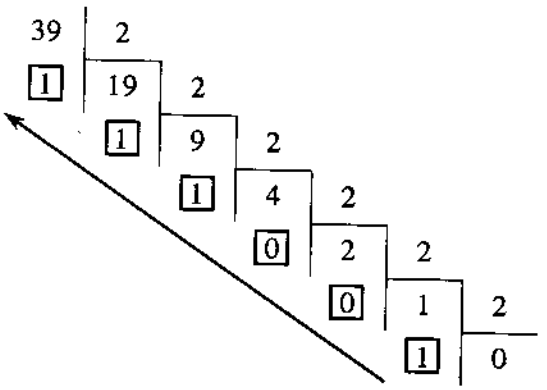
Nếu đem con số này chia cho 10 và để ý tới các số dư ta sẽ thấy :



Mã số 274 chính là dạng hiển thị của các số dư, theo thứ tự xuất hiện ngược lại.

Tương tự như vậy, mã số nhị phân của một con số, cho dưới dạng thập phân, cũng sẽ được xác định với phép chia và lấy số dư, chỉ có khác là bây giờ thực hiện phép chia với 2 và dư số sẽ là số 0 hoặc số 1 thôi. Ví dụ :

a)

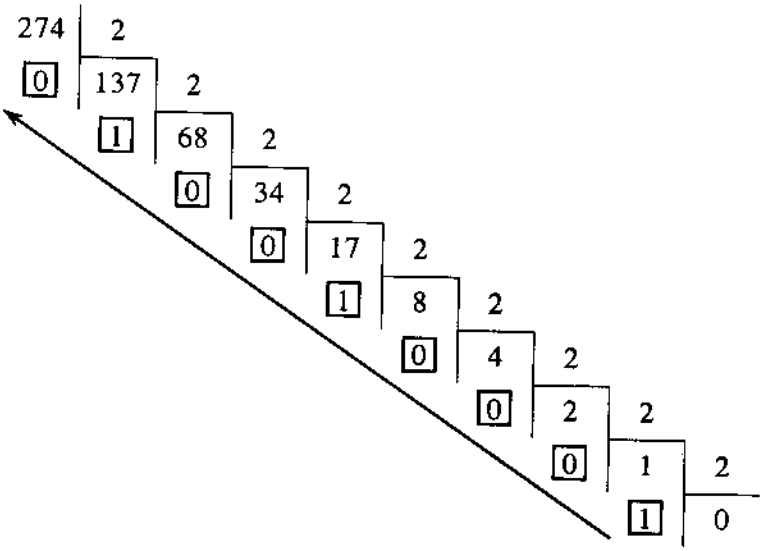


Ta sẽ có : mã số nhị phân biểu diễn cho số 39 : 100111.

Ta có thể viết :

$$(39)_{10} = (100111)_2$$

b)



Vậy $(274)_{10} = (100010010)_2$

Như vậy rõ ràng là trong cách chuyển đổi này các số dư được tạo ra sau lại được hiển thị trước. Cơ chế sắp xếp này rất phù hợp với cơ chế hoạt động của stack. Vì vậy trong giải thuật chuyển đổi số nguyên từ thập phân sang nhị phân, người ta sử dụng một stack để lưu trữ các số dư, sau đó lại lấy các số này từ stack ra để hiển thị thành mã số nhị phân.

Giả sử stack này được cài đặt bởi một vectơ lưu trữ S, mà T là con trỏ, trỏ tới đỉnh ; lúc đầu stack rỗng, nghĩa là T = 0.

Có thể viết giải thuật chuyển đổi một số nguyên N sang dạng nhị phân bằng thủ tục sau :

Procedure CHANGE (N) ;

1. m := N ;

2. {Tính số dư và nạp vào stack S}

while m ≠ 0 **do begin**

R := m mod 2 ; {Tính số dư}

call PUSH (S, T, R) ; {nạp R vào stack S}

m := m div 2 {thay m bằng thương của phép chia m cho 2}

end ;

3. {Hiển thị từng chữ số nhị phân trong mã số biểu diễn N}

while T ≠ 0 **do begin**

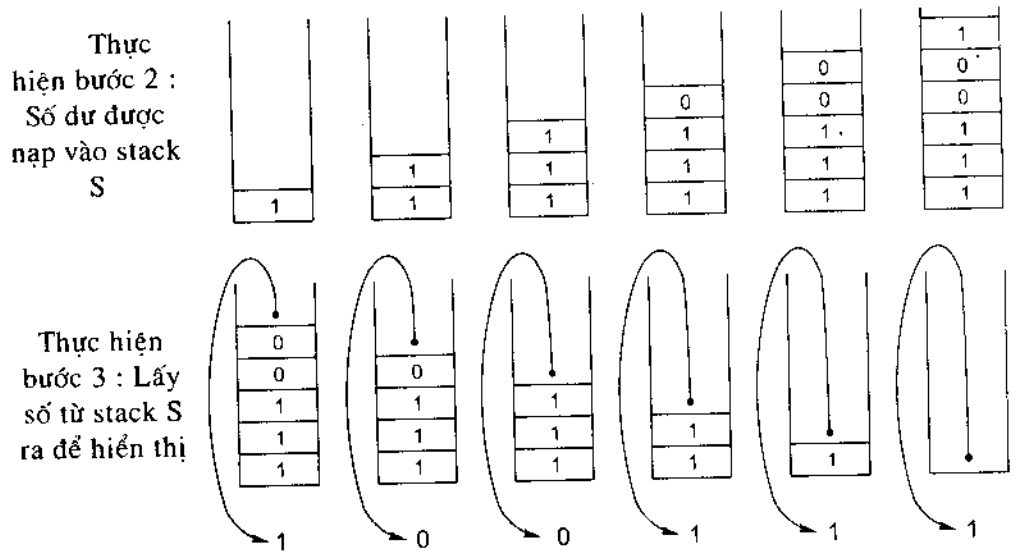
call POP (S, T, ,X) ; {lấy số ra khỏi stack}

write (X)

end ;

4. **return**

Nếu N = 39, như ở ví dụ trên, thì tình trạng của stack S, khi thực hiện thủ tục CHANGE (N) được minh họa như sau



HÌNH 4.3

4.3.2. Biểu thức số học với kí pháp BA – LAN

Ta xét một biểu thức số học với các phép toán hai ngôi như các phép : cộng (+), trừ (–), nhân (*), chia (/), lũy thừa (↑) v.v...

Ta thấy, với phép toán này thì toán tử (dấu phép toán) bao giờ cũng đặt ở giữa hai toán hạng (vì vậy ta nói : biểu thức số học thông thường được viết theo *kí pháp trung tố* (infix notation). Với kí pháp này thì nhiều khi để phân biệt hai toán hạng ứng với một toán tử ta bắt buộc phải dùng các cặp dấu ngoặc đơn hoặc nếu không thì phải chấp nhận một thứ tự ưu tiên giữa các phép toán, như đã được quy định trong các ngôn ngữ lập trình. Cụ thể là thứ tự ưu tiên được xếp theo trình tự sau :

1. Phép lũy thừa
2. Phép nhân, phép chia
3. Phép cộng, phép trừ

Các phép toán cùng thứ tự ưu tiên thì sẽ được thực hiện theo trình tự : trái trước, phải sau (trong biểu thức)

Ví dụ : $A + B * C - D$

sẽ được thực theo trình tự :

$B * C$

rồi $A + (B * C)$

và cuối cùng là : $(A + (B * C)) - D$

tương tự như vậy

$A * B \uparrow 2 - C / D + E$

sẽ được thực hiện :

$B \uparrow 2$

rồi : $A * (B \uparrow 2)$

C / D

$A * (B \uparrow 2) - (C / D)$

cuối cùng là : $(A * (B \uparrow 2) - (C / D)) + E$

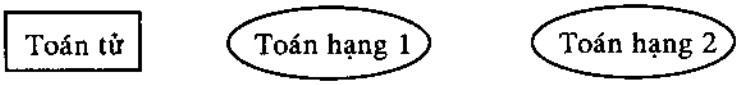
(Chú ý là : dấu ngoặc trong cách viết trên tạ thêm vào để phân biệt toán hạng ứng với một toán tử).

Rõ ràng cách viết biểu thức theo kí pháp trung tố với việc sử dụng dấu ngoặc và thứ tự ưu tiên giữa các phép toán đã khiến cho việc tính toán giá trị biểu thức khá "cồng kềnh".

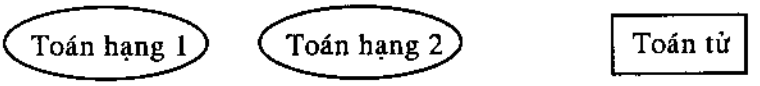
Trong những năm đầu 1950, nhà toán học Ba Lan : Jan Lukasiewicz đã đưa ra dạng kí pháp mới để biểu diễn các biểu thức mà không cần tới dấu ngoặc, đó là kí pháp *tiền tố* (prefix notation) và *hậu tố* (postfix notation) mà gọi chung là *kí pháp Ba – Lan* (Polish notation)

1. Biểu thức dạng tiền tố và hậu tố

Với kí pháp tiền tố thì toán tử được đặt trước toán hạng 1 và toán hạng 2, nghĩa là theo mô hình



Còn hậu tố thì nó lại được đặt sau, nghĩa là :



Ta có thể thấy cụ thể qua các ví dụ sau

Ví dụ 1 :

Dạng trung tố	Dạng tiền tố	Dạng hậu tố
A + B	+ A B	A B +
A / B	/ A B	A B /
(A + B) * C	* + A B C	A B + C *
(A + B) / (C – D)	/ + A B – C D	A B + C D – /
A + B / C – D	– + A / B C D	A B C / + D –

Ta có thể biến đổi "trực tiếp" từ biểu thức dạng trung tố sang tiền tố hoặc hậu tố nếu dựa vào mô hình quy tắc như đã nêu trên và ứng với mỗi toán tử ta xác định được đâu là toán hạng 1, đâu là toán hạng 2 (mỗi toán hạng lại có thể là một biểu thức và ta lại xác định tương tự).

Ví dụ 2 :

a) Với biểu thức (A + B) / (C – D) thì ứng với phép chia /, (A + B) là toán hạng 1 còn (C – D) là toán hạng 2. Vậy thì biểu thức tiền tố có dạng : / (A + B)(C – D). Nhưng A + B lại là một biểu thức mà dạng tiền tố là + A B. Còn C – D thì dạng tiền tố là – C D.

Tóm lại : biểu thức tiền tố ứng với (A + B) / (C – D) sẽ có dạng / + A B – C D, như đã nêu.

b) Với biểu thức A + B / C – D thì nó tương đương như (A + (B / C)) – D nên biểu thức hậu tố sẽ có dạng : (A + (B / C))D –

mà A + (B / C) lại có dạng A(B / C) +

và B / C thì thay bằng B C /

Tóm lại : biểu thức hậu tố của A + B / C – D có dạng A B C / + D –

Chú ý. Trong các ngôn ngữ lập trình, các biểu thức số học thường được viết theo kí pháp trung tố. Chương trình dịch của ngôn ngữ sẽ chuyển các biểu thức này sang dạng hậu tố (hoặc tiền tố), sau đó việc tính giá trị của biểu thức sẽ được thực hiện trên dạng hậu tố này.

Dĩ nhiên việc chuyển một biểu thức từ dạng trung tố sang hậu tố phải được thực hiện bởi một chương trình. Ở đây ta không xét tới chương trình đó. Việc biến đổi biểu thức từ trung tố sang hậu tố (hoặc tiền tố) trong phần bài tập, chỉ là phần áp dụng quy tắc biến đổi trực tiếp như đã nêu ở trên thôi.

2. Tính giá trị của một biểu thức dạng hậu tố

Trước hết, để cho đơn giản, ta giả sử rằng trong các biểu thức xét ở đây tên biến chỉ gồm một chữ cái và hằng chỉ là một chữ số thôi. Như vậy thì một biểu thức dạng hậu tố là một xâu kí tự mà mỗi kí tự là ứng với một biến, hằng hoặc phép toán.

Chẳng hạn, biểu thức hậu tố :

A B + C – D E * /

thì A, B, C, D, E là các biến, còn các phép toán là +, –, *, /.

Nếu đọc biểu thức hậu tố từ trái qua phải ta sẽ thấy khi một toán tử xuất hiện thì 2 toán hạng vừa đọc sát nó sẽ được kết hợp với toán tử này để tạo thành toán hạng mới ứng với toán tử sẽ được đọc sau đó, và cứ như vậy. Với biểu thức vừa nêu ở trên khi gặp toán tử cộng (+) thì 2 toán hạng A và B được kết hợp với nó, nghĩa là (A + B). Khi đọc toán tử trừ (–) thì hai toán hạng ở trước và sát nó, cụ thể là (A + B) và C sẽ được kết hợp lại để có (A + B) – C. Khi gặp toán tử nhân (*) ta sẽ có D * E. Khi gặp toán tử chia (/) thì sẽ thành ((A + B) – C) / (D * E).

Việc tính giá trị của biểu thức theo cách như trên sẽ được thực hiện một cách rất "máy móc", không phải băn khoăn gì về chuyện dấu ngoặc hay thứ tự ưu tiên của phép toán nữa.

Giả sử trước khi tính các biến đã có giá trị như sau : A = 5 ; B = 9 ; C = 2 ; D = 2 ; E = 3 thì:

Khi đọc phép : +, sẽ thực hiện 5 + 9 = 14

Khi đọc phép : –, sẽ thực hiện 14 – 2 = 12

Khi đọc phép : *, sẽ thực hiện 2 * 3 = 6

Khi đọc phép : /, sẽ thực hiện 12 / 6 = 2

Xét thêm một ví dụ nữa :

A B C D + – *

Giả sử lúc đó A = 2, B = 9, C = 4, D = 3

Khi đọc phép + thì sẽ thực hiện C + D và cho giá trị là 4 + 3 = 7

Khi đọc phép – thì sẽ thực hiện B – (C + D) và cho giá trị 9 – 7 = 2

Khi đọc phép * thì sẽ thực hiện A * (B – (C + D))

và cho giá trị 2 * 2 = 4

Nếu chú ý ta sẽ thấy :

• Trước khi đọc tới toán tử thì giá trị của các toán hạng phải được bảo lưu để chờ thực hiện phép tính.

• Hai toán hạng được đọc sau thì lại được kết hợp với toán tử đọc trước, điều đó cũng có nghĩa là hai giá trị được bảo lưu sau thì lại phải lấy ra trước để tính toán. Điều này trùng khớp với cơ chế "vào – sau – ra – trước" của stack. Vì vậy : để xác định giá trị của một biểu thức hậu tố (ứng với các giá trị của biến và hằng trong biểu thức đó) người ta cần dùng một stack để bảo lưu các giá trị của toán hạng.

Cụ thể, cách tính có thể tóm tắt như sau :

* Đọc biểu thức hậu tố từ trái sang phải

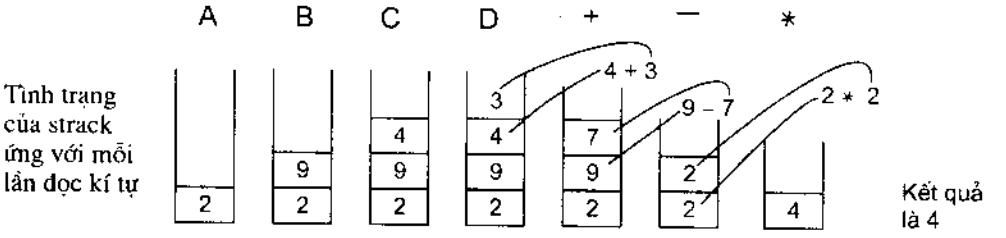
– Nếu kí tự được đọc X là toán hạng thì nạp giá trị của X vào stack.

– Nếu kí tự X là toán tử thì : Lần lượt lấy từ stack ra 2 giá trị

Tác động phép toán X giữa giá trị lấy ra sau với giá trị lấy ra trước rồi nạp kết quả vào stack.

Quá trình trên được tiếp tục cho tới khi kết thúc biểu thức. Lúc đó giá trị còn trong stack chính là giá trị của biểu thức.

Có thể minh họa cách tính này với ví dụ vừa nêu :



HÌNH 4.4

Sau đây là giải thuật phản ảnh cách tính giá trị của biểu thức hậu tố, như đã nêu

Procedure EVAL (P, VAL)

{giải thuật này thực hiện tính giá trị của biểu thức hậu tố P, tương ứng với giá trị của các biến, kết quả sẽ được gán cho VAL.

Trong giải thuật có dùng một stack S với T trỏ tới đỉnh, thoạt đầu thì T = 0 (stack rỗng)}

1. Ghi thêm dấu ")" vào cuối xâu P để làm dấu kết thúc xâu ;

2. **repeat**

Đọc kí tự X trong P, khi duyệt từ trái sang phải ;

3. **If** X là một toán hạng **then call** PUSH (S, T, X)

else begin

4. **call** POP (S, T, Y)

call POP (S, T, Z) ;

W = Z Ⓧ Y ;

{Ⓧ chỉ phép toán X}

call PUSH (S, T, W)

end ;

until gặp dấu kết thúc xâu ")" ;

5. **call** POP (S, T, VAL) ;

6. **return**

Chú ý : Stack còn có một vai trò rất quan trọng trong việc cài đặt các thủ tục đệ quy hay nói một cách khác : nó là công cụ để "khử đệ quy". Tuy nhiên, ta sẽ không đi sâu vào phần này.

4.4. ĐỊNH NGHĨA QUEUE

Khác với stack, queue là một kiểu danh sách đặc biệt mà phép bổ sung thực hiện ở một đầu, gọi là *lối sau* (rear) còn phép loại bỏ lại thực hiện ở một đầu khác, gọi là *lối trước* (front).

Cơ cấu của queue giống như một hàng đợi : hàng người chờ lĩnh tiền ở quầy tiết kiệm, hàng ô tô chờ qua phà, danh sách khách hàng đăng kí mua vé cho một chuyến bay v.v..

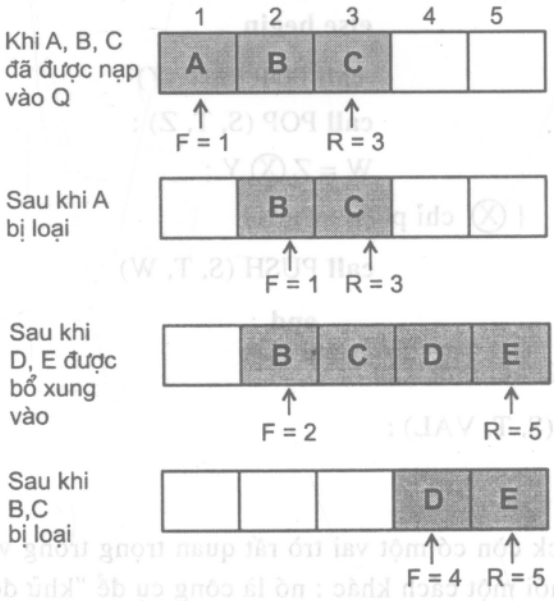
Tất nhiên các phần tử của queue phải được xử lí theo thứ tự mà chúng được tạo ra, nghĩa là vào ở đầu này thì ra ở đầu kia và vào trước thì ra trước. Vì vậy queue còn được gọi là danh sách kiểu FIFO (first – in – first – out).

4.5. LƯU TRỮ KẾ TIẾP ĐỐI VỚI QUEUE

Tương tự như stack, người ta có thể dùng một vectơ lưu trữ Q có n phần tử làm cấu trúc lưu trữ cho queue.

Rõ ràng là ta phải nắm được hai biến trỏ : R trỏ tới lối sau và F trỏ tới lối trước. (Chú ý là ở đây R ghi nhận giá trị là chỉ số ứng với phần tử vừa được bổ sung vào ở lối sau, F ghi nhận chỉ số ứng với phần tử sẽ bị loại bỏ, đang ở lối trước). Khi queue rỗng thì F = R = 0. Khi bổ sung một phần tử mới vào, R sẽ tăng lên. Nhưng khi loại bỏ một phần tử đi thì F cũng tăng lên.

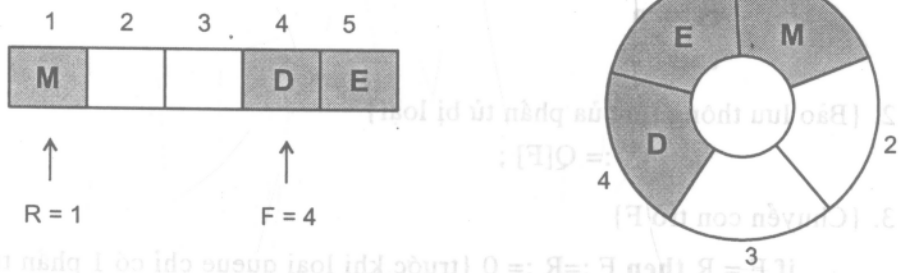
Sau đây là một vài tình huống ứng với Q có 5 phần tử



HÌNH 4.5

Nếu bây giờ bổ sung thêm một phần tử mới vào (giả sử phần tử M) thì không thể lại tăng R = 6 được, vì như vậy sẽ TRÀN ra ngoài phạm vi lưu trữ của Q, mà thực ra vectơ lưu trữ Q vẫn còn 3 "chỗ trống" !

Khó khăn này có thể khắc phục được nếu như ta hình dung Q như có cấu trúc vòng tròn, nghĩa là Q[1] được coi như đứng sau Q[5] và phần tử mới M sẽ được bổ sung vào Q[1]. Có thể minh họa như sau :



HÌNH 4.6

Bằng cách này việc bổ sung sẽ không thể thực hiện được chỉ khi vectơ lưu trữ Q của queue đã thực sự đầy.

Sau đây là giải thuật thực hiện phép bổ sung và phép loại bỏ vào queue, được cài đặt bởi vectơ Q, có n phần tử và có cấu trúc vòng tròn như đã nêu.

Procedure QINSERT (Q, F, R, X) ;

{ ở đây X là thông tin ứng với phần tử mới sẽ được bổ sung vào queue}

1. {Q đã đầy}

if F = 1 **and** R = n or F = R + 1 **then begin**

write ('queue sẽ TRÀN') ;

return

end ;

2. {chuyển con trỏ R}

if F = 0 **then** F : = R : = 1 {trước khi bổ sung queue còn rỗng}

else if R = n **then** R := 1

else R := R + 1 ;

3. {Bổ sung X vào}

Q[R] := X ;

4. **return**

Procedure QDELETE (Q, F, R, Y) ;

{giải thuật này sẽ bảo lưu thông tin của phần tử bị loại và ghi nhận bởi Y}

1. {Q đã rỗng}

if F = 0 **then begin**

write ('Queue đã CẠN') ;

return

end ;

2. {Bảo lưu thông tin của phần tử bị loại}

Y := Q[F] ;

3. {Chuyển con trỏ F}

if F = R **then** F :=R := 0 {trước khi loại queue chỉ có 1 phần tử}

else if F = n **then** F :=1

else F := F + 1

4. **return** ;

Chú ý : Queue được dùng một cách phổ biến để thực hiện các "tuyến chờ" (waiting lines) trong các xử lí động, đặc biệt trong các hệ mô phỏng. Ví dụ : nhiều khách hàng cùng đăng kí mua vé cho một chuyến bay. Họ sẽ phải "xếp hàng" để chờ đến lượt. Chương trình máy tính mô phỏng quá trình "xếp hàng" phải lưu trữ các thông tin cần thiết về khách hàng vào trong queue để xử lí theo kiểu "đăng kí trước thì được phục vụ trước"

4.6. LƯU TRỮ MÓC NỐI VỚI STACK VÀ QUEUE

Như ta đã biết, đối với stack việc truy cập chỉ thực hiện ở một đầu (đỉnh). Vì vậy việc cài đặt stack bằng một danh sách nối đơn, có con trỏ L trỏ tới nút đầu tiên, là một điều khá tự nhiên. Ta có thể coi L như con trỏ đang trỏ tới đỉnh stack. Bổ sung một nút vào stack chính là bổ sung một nút vào để nó trở thành nút đầu tiên của danh sách, loại bỏ một nút ra khỏi stack chính là loại bỏ nút đầu tiên của danh sách, đang trỏ bởi L. Hai giải thuật tương ứng với hai phép bổ sung và loại bỏ này được diễn đạt bởi các thủ tục sau :

Procedure PUSH – STACK (L, X)

{L là con trỏ, trỏ tới đỉnh stack móc nối, (nút đầu tiên) ; X là thông tin cho nút mới để bổ sung}

1. {Tạo lập nút mới}

call New (p) ;

INFO (p) := X ;

2. {bổ sung}

LINK (p) :=L ;

L :=p

3. **return**

Chú ý. Trong thủ tục PUSH, đối với stack cài đặt bằng mảng (vectơ lưu trữ S) ta phải kiểm tra hiện tượng stack đã đầy (vì nếu bổ sung thì stack sẽ TRÀN). Sở dĩ như vậy, vì kích thước của vectơ lưu trữ bị hạn chế. Còn đối với stack móc nối thì kích thước của danh sách nối đơn chỉ bị hạn chế khi bộ nhớ không còn một nút trống nào, mà điều này thường ít xảy ra, nên ở đây ta không phải thực hiện việc kiểm tra này.

Procedure POP – STACK (L, Y) ;

{Thủ tục này thực hiện loại bỏ một nút ra khỏi stack móc nối, có con trỏ L đang trỏ tới đỉnh. Thông tin của nút bị loại được bảo lưu bởi Y}

1. {Trường hợp stack rỗng}

if L = null **then begin**

write ('stack rỗng') ;

return

end ;

2. {Bảo lưu thông tin}

Y :=INFO (L) ;

3. {Loại bỏ}

T :=L ;

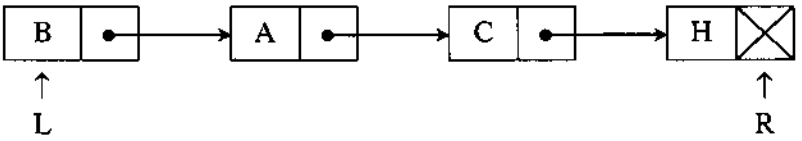
L := LINK (L) ;

call *dispose* (T) ;

4. **return**

• Đối với queue thì loại bỏ thực hiện ở một đầu còn bổ sung lại ở đầu khác. Nếu cho danh sách móc nối hoạt động như một queue và coi L trỏ tới lối trước thì việc loại bỏ một nút sẽ không khác gì như với stack (thủ tục như

POP – STACK), nhưng việc bổ sung đòi hỏi phải duyệt qua danh sách để tìm địa chỉ nút cuối cùng nghĩa là xác định được "lối sau". Có thể tránh được điều này nếu như ngay từ đầu ta nắm được hai con trỏ ; L trỏ tới nút đầu tiên (lối trước) và R trỏ tới nút cuối cùng (lối sau). Khi queue rỗng thì L = R = null.



Với điều kiện như trên thì phép bổ sung phần tử mới vào queue chính là phép bổ sung nút mới để nó trở thành nút cuối cùng của danh sách.

Sau đây là giải thuật :

Procedure INSERT – QUEUE (L, R, X) ;

1. {Tạo lập nút mới}

call New (p) ;

INFO (p) := X ; LINK (p) : = **null** ;

2. {Trường hợp queue rỗng trước khi bổ sung}

if L = null **and** R = null **then begin**

L := R := p

return

end ;

3. LINK (R) := p ;

R :=p

4. **return**

Câu hỏi và bài tập

4.1. Anh (chị) hãy nêu vài ví dụ thực tế mà ở đó có cơ chế hoạt động, "vào sau ra trước".

4.2. Theo ý anh (chị), đối với stack và queue thì có những cách cài đặt nào ? Hãy so sánh đặc điểm của các cách đó.

4.3. Hãy nêu các bước thực hiện tính giá trị của một biểu thức tiền tố và nhận xét những chỗ khác nhau so với cách tính giá trị của biểu thức hậu tố (như đã nêu trong bài).

4.4. Cho một danh sách nối đơn có con trỏ L trỏ tới nút đầu tiên, con trỏ R trỏ tới nút cuối cùng. Có thể cho nó hoạt động như một queue, với R trỏ tới lối trước và L trỏ tới lối sau, được không ? Lúc đó phép bổ sung và phép loại bỏ thực hiện thế nào ?

4.5. Cho một stack, được cài đặt bởi một vectơ S có n = 6 phần tử và có con trỏ T trỏ tới đỉnh. Thoạt đầu stack rỗng : T = 0.

Hãy xác định kết quả cuối cùng khi thực hiện các phép tính sau :

1. A :=2 ; B :=5 ;
2. **call** PUSH (S, T, A) ;

   call PUSH (S, T, 4) ;

   call PUSH (S, T, B + 2) ;

   call PUSH (S, T, 9) ;

   call PUSH (S, T, A + B) ;
3. **while** T ≠ 0 **do begin**

   call POP (S, T, X) ;

   write (X) ;
4. **return.**

4.6. Hãy minh họa tình trạng của stack, qua các bước thực hiện giải thuật CHANGE để biến đổi các số sau sang dạng nhị phân

a) 69

b) 531

4.7. Hãy biến đổi các biểu thức sau đây sang dạng tiền tố, hậu tố :

a) (x + y) * z

b) x – y – z * (A + B)

c) A + B * C ↑ D / E – F

d) ((A + B) * D) ↑ (E – F)

4.8. Hãy biến đổi sang dạng trung tố

1. Các biểu thức tiền tố sau :

   a) * + A B – C D

   b) / + A * B C – D E
2. Các biểu thức hậu tố sau :

   a) A B C + – D *

   b) A B + C – D E * /

4.9. Giả sử A = 12.0 ; B = 7.0 ; C = 3.0 ; d = 2.0 ; E = 1.0

Hãy tính giá trị của các biểu thức hậu tố đã nêu trong câu 2 bài tập 4.8.

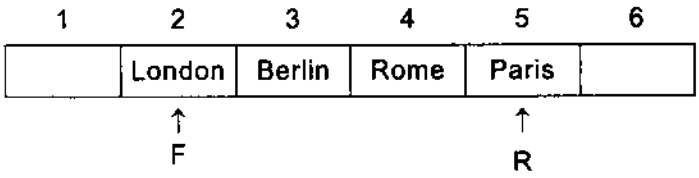
4.10. Cho biểu thức hậu tố P

P : A B C – / D E F + * +

Hãy minh họa tình trạng của stack qua các bước thực hiện giải thuật EVAL. Để tính giá trị của P, ứng với A = 8.0 ; B = 5.0 ; C = 3.0 ; D = 2.0 ; E = 7.0 ; F = 1.0

4.11. Cho một queue được lưu trữ trong bộ nhớ bởi một vectơ Q có n = 6 phần tử (ô nhớ), được hoạt động theo cấu trúc vòng tròn.

Thoạt đầu Q có dạng



Hãy minh họa tình trạng của Q và nêu rõ giá trị tương ứng của F và R sau mỗi lần thực hiện các phép dưới đây :

a) "Madrid" được bổ sung vào queue

b) Loại tên hai thành phố ra khỏi queue

c) "Oslo" được bổ sung vào queue

d) "Moscow" được bổ sung vào queue

e) Loại tên ba thành phố ra khỏi queue.

4.12. Cho một queue được cài đặt bởi vectơ Q có n = 12 phần tử, hoạt động theo cấu trúc vòng tròn. Hãy xác định số phần tử của queue nếu :

a) F = 4 ; R = 8

b) F = 10 ; R = 3

c) F = 5 ; R = 6.

Chương 5. Cấu trúc cây (tree)

5.1. ĐỊNH NGHĨA VÀ MỘT SỐ KHÁI NIỆM

Cây là một cấu trúc phi tuyến, thiết lập trên một tập hữu hạn các phần tử mà ta gọi là "nút", trong đó có một nút đặc biệt được gọi là gốc (root), liên kết bởi một quan hệ phân cấp, gọi là quan hệ cha – con.

Cây có thể được định nghĩa một cách đệ quy như sau :

1. Một nút là một cây. Nút đó cũng là gốc của cây ấy.

2. Nếu T_1, T_2, ..., T_k là các cây với n_1, n_2, ..., n_k lần lượt là các gốc ; n là một nút và n có quan hệ cha – con với n_1, n_2, ..., n_k thì lúc đó một cây mới T sẽ được tạo lập, với n là gốc của nó. Nút n được gọi là *cha* của n_1, n_2, ..., n_k ; ngược lại n_1, n_2, ..., n_k được gọi là *con* của n. Các cây T_1, T_2, ..., T_k được gọi là *cây con* (subtrees) của n.

Người ta quy ước : một cây không có nút nào được gọi là *cây rỗng*.

Trên hình vẽ, người ta biểu diễn cây với nút gốc ở trên và quan hệ cha – con được thể hiện bởi một đoạn thẳng (giữa nút cha và nút con).

Ví dụ : Chương 1 của giáo trình này có cấu trúc cây.

1. Giải thuật

    1.1. Cấu trúc dữ liệu và giải thuật

    1.2. Ngôn ngữ diễn đạt giải thuật

    1.3. Thiết kế giải thuật

    1.4. Đánh giá giải thuật

        1.4.1. Đặt vấn đề

        1.4.2. Thời gian thực hiện trung bình

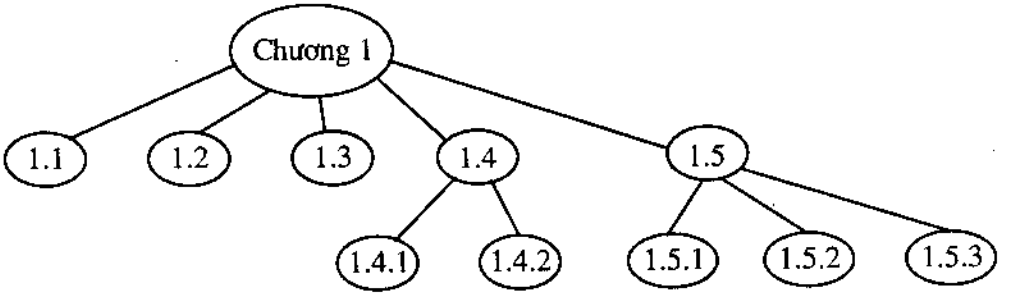
    1.5. Giải thuật đệ quy

        1.5.1. Định nghĩa

        1.5.2. Ví dụ về thủ tục đệ quy

        1.5.3. Chú ý

Cây này có thể biểu diễn như hình 5.1.

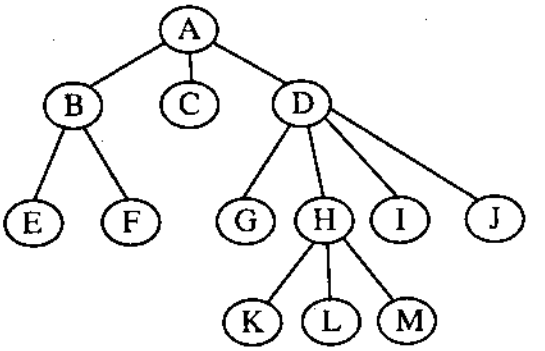


HÌNH 5.1.

Sau đây là một số khái niệm :

a) Số các con của một nút được gọi là *cấp* (degree) của nút đó. Nút có cấp bằng không được gọi là *lá* (leaf) hay *nút tận cùng* (terminal node). Nút không phải là lá được gọi là *nút nhánh* (branch node).

Cấp cao nhất của nút trên cây gọi là *cấp của cây* đó. Ví dụ : với cây ở hình 5.2 (ở đây các chữ A, B, C,... tượng. trưng cho phần thông tin (dữ liệu) ứng với mỗi nút).



HÌNH 5.2

A là gốc ; B, C, D là *con* của A ; D là *cha* của G, H, I, J ; A có cấp bằng 3, D có *cấp* bằng 4. Các nút như E, F, C, G, K, ... là *lá*. Các nút như B, D, H... là các *nút nhánh*. Cây trên có cấp bằng 4.

b) Gốc của cây có *mức* (level) bằng 1.

Nếu nút cha có mức là i thì nút con có mức là i + 1.

Như ở cây trên :

A có mức là 1 ;

B, C, D có mức là 2 ;

E, F, G, I, J có mức là 3 ;

K, L, M có mức là 4.

c) *Chiều cao* (height) hay *chiều sâu* (depth) của một cây là số mức lớn nhất của nút có trên cây đó.

Cây ở hình 5.1 có chiều cao là 3

Cây ở hình 5.2 có chiều cao là 4.

d) Nếu n_1, n_2, ..., n_k là dãy các nút mà n_i là cha của n_{i+1} với $1 \le i < k$ thì dãy đó được gọi là *đường đi* (path) từ n_1 đến n_k.

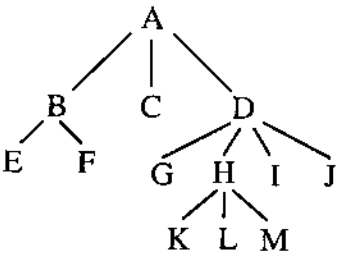
Độ dài của đường đi (path length) bằng số nút trên đường đi đó trừ đi 1. Ví dụ với cây ở hình 5.2 : độ dài đường đi từ A đến L bằng 3, độ dài đường đi từ D tới M bằng 2.

e) Nếu thứ tự các cây con của một nút được coi trọng thì cây đang xét là *cây có thứ tự* (ordered tree), ngược lại là *cây không có thứ tự* (unordered tree).

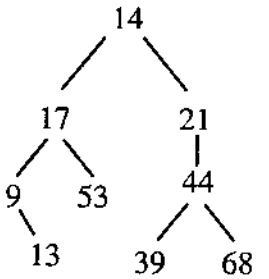
f) Đối với nút trên cây ngoài khái niệm cha, con, người ta còn có thể mở rộng sang các khái niệm khác theo quan hệ như trong gia tộc. Ví dụ như trong hình 5.2. A, D, H... được gọi là *tiền bối* của L ; còn E, G, K... được gọi là *hậu sinh* của A... ; các nút G, H, I, J là các nút *anh em* v.v..

Chú ý. Đôi lúc, để cho tiện, hình vẽ minh họa của cây sẽ được thể hiện đơn giản : nút trên cây chỉ tượng trưng bằng chữ hoặc số.

Ví dụ cây ở hình 5.2, có thể minh họa bởi hình 5.3. Hoặc cây mà mỗi nút đều chứa một số như hình 5.4.



HÌNH 5.3



HÌNH 5.4

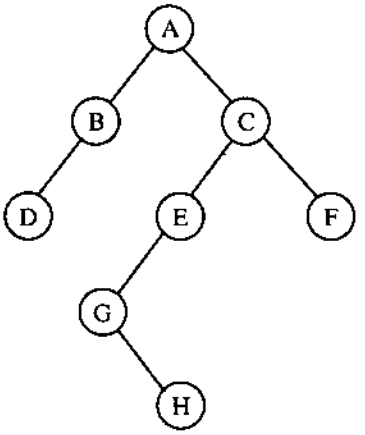
5.2. CÂY NHỊ PHÂN

5.2.1. Định nghĩa

Cây nhị phân là một dạng quan trọng của cấu trúc cây.

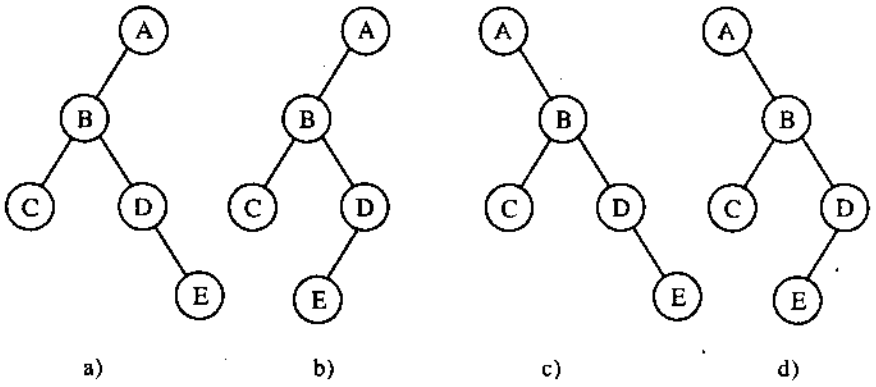
Nó có đặc điểm là : mọi nút trên cây chỉ có tối đa là hai con. Đối với cây con của mỗi nút thì cũng phân biệt *cây con trái* (left subtree) và *cây con phải* (right subtree).

Như vậy cây nhị phân là một cây có thứ tự. Ví dụ : cây trên hình 5.5 là một cây nhị phân có A là nút gốc, cây con trái của A có gốc là B, cây con phải của A có gốc là C.



HÌNH 5.5

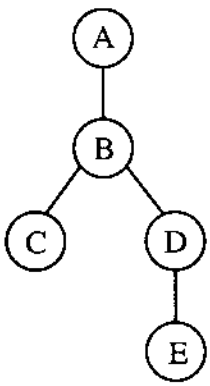
Các cây trên hình 5.6 là các cây nhị phân khác nhau.



HÌNH 5.6

Chẳng hạn : cây a) khác cây b) vì với a) (E) là con phải của (D) còn với b) thì (E) lại là con trái của (D) cây a) khác cây c) vì với c) (B) không phải là con trái mà là con phải của (A) ...

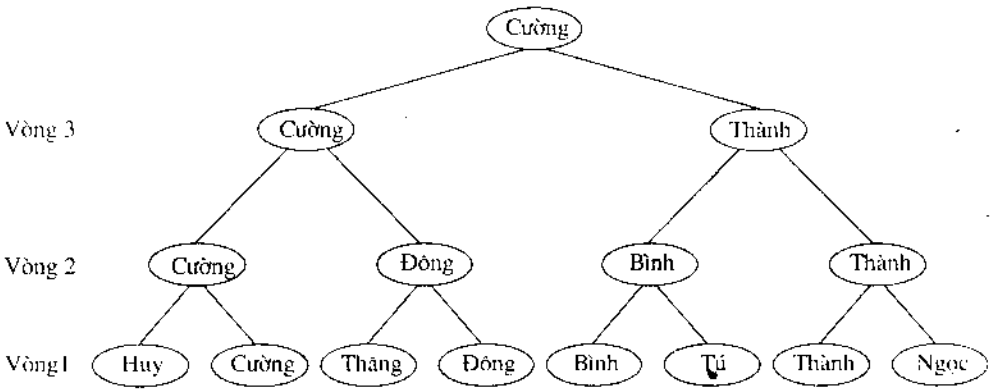
Nếu không để ý tới thứ tự của cây con thì cả bốn cây nêu trên đều chỉ là một, mà có thể minh họa bởi hình 5.7 :



HÌNH 5.7

Có rất nhiều đối tượng có thể biểu diễn theo cấu trúc cây nhị phân, chẳng hạn :

– Kết quả thi đấu bóng bàn của 8 đối thủ được tuyển chọn theo từng cặp, ai thắng sẽ được chọn vào vòng sau, có thể biểu diễn như hình 5.8.



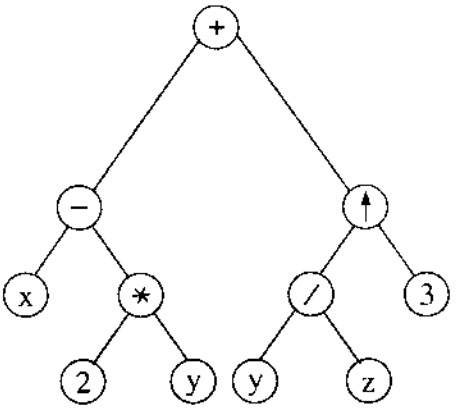
HÌNH 5.8

– Biểu thức số học với phép toán hai ngôi nếu coi toán tử là ứng với gốc, toán hạng 1 ứng với cây con trái, toán hạng 2 ứng với cây con phải, thì có thể biểu diễn bởi cây nhị phân.

Chẳng hạn :

$$(x - 2 * y) + (y / z) \uparrow 3$$

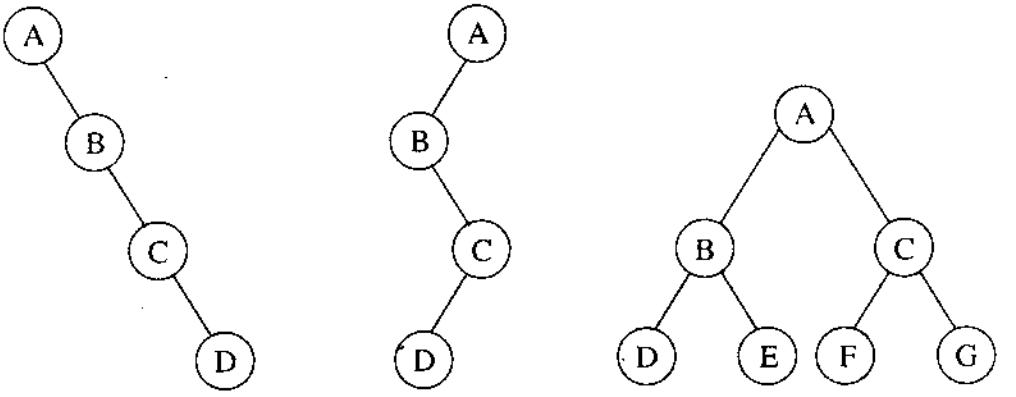
có thể biểu diễn như sau :



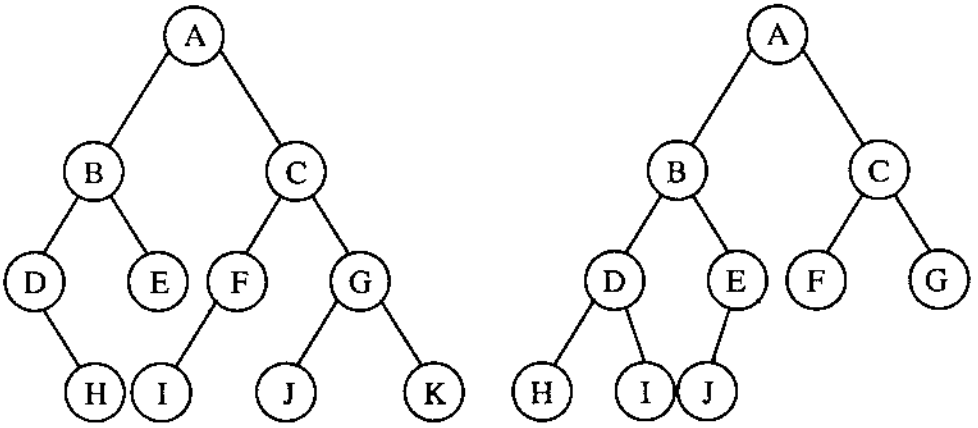
HÌNH 5.9

Cần chú ý tới một số dạng đặc biệt của cây nhị phân tương tự như ở hình 5.10.

Các cây như hình 5.10a,b được gọi là *cây suy biến* (degenerate binary tree) trên đó, trừ nút lá, các nút nhánh đều chỉ có 1 con.



HÌNH 5.10a, b, c)



HÌNH 5.10d,e)

Cây ở hình 5.10c được gọi là *cây đầy đủ* (full tree), trên đó trừ nút lá, các nút nhánh đều có 2 con hay nói một cách khác : số nút ở mọi mức trên cây đều "đầy đủ" cả.

Cây ở hình 5.10d có số nút đầy đủ ở mọi mức, trừ ở mức cuối cùng. Ta sẽ gọi là *cây gần đầy*.

Nếu cây gần đầy mà ở mức cuối cùng các nút đều dạt về phía trái, như cây hình 5.10e thì được gọi là *cây hoàn chỉnh*.

5.2.2. Tính chất

Bây giờ ta sẽ xét tới một vài tính chất đặc biệt của cây nhị phân, qua bổ đề sau đây :

Bổ đề :

1) Số lượng tối đa của các nút ở mức i trên cây nhị phân là 2^{i-1} ($i \geq 1$).

2) Số lượng tối đa các nút trên một cây nhị phân có chiều cao h là 2^h-1 ($h \geq 1$).

Chứng minh :

1) Ta sẽ chứng minh sự đúng đắn của mệnh đề này bằng quy nạp toán học :

Ta thấy : ở mức 1 ($i = 1$) cây nhị phân có tối đa là 1 nút = 2^0 nút = 2^{1-1} nút, như vậy là mệnh đề đã đúng.

Giả sử mệnh đề đã đúng với mức $i = k$, nghĩa là : số lượng tối đa các nút ở mức k trên cây nhị phân là 2^{k-1}.

Xét ở mức $k + 1$; để có số lượng nút tối đa ở mức này thì mỗi nút ở mức k phải có 2 con, mà ở mức này, như đã giả thiết, ta có 2^{k-1} nút. Vậy ở mức k sẽ

có tối đa là (2^{k-1}.2) nút hay 2^k nút = $2^{(k+1)-1}$ nút. Mệnh đề đã đúng với mức (k + 1).

Ta đã kiểm chứng : mệnh đề đã đúng ở mức 1 vậy nó sẽ đúng ở mức 2 và như thế thì sẽ đúng ở mức 3, mức 4, v.v... nghĩa là nó đúng với mọi mức $i \geq 1$.

2) Theo định nghĩa chiều cao của cây là số mức lớn nhất ứng với các nút có trên cây đó. Cây có chiều cao h thì số mức cao nhất ứng với cây đó cũng là h.

Theo 1) ta suy ra số nút tối đa có trên cây nhị phân chiều cao h, nghĩa là tổng số nút tối đa có ở các mức từ 1 tới h, sẽ bằng :

$$2^0 + 2^1 + 2^2 + ... + 2^{h-1} = \frac{2^h - 1}{2 - 1} = 2^h - 1$$

(tổng của cấp số nhân có số hạng đầu tiên bằng 1 và số công bội bằng 2).

• Từ kết quả trên có thể suy ra :

1. Nếu cây nhị phân đầy đủ có chiều cao h và số nút là n thì

$$n = 2^h - 1$$

hay $2^h = n + 1$

Vậy $h = \log_2(n + 1)$

2. Nếu cây nhị phân gần đầy có chiều cao là h, có số nút là n thì

$$2^{(h-1)} - 1 < n < 2^h - 1$$

Do đó : $h = \lceil \log_2(n + 1) \rceil$

(kí hiệu $\lceil x \rceil$ chỉ số nguyên trên của x.

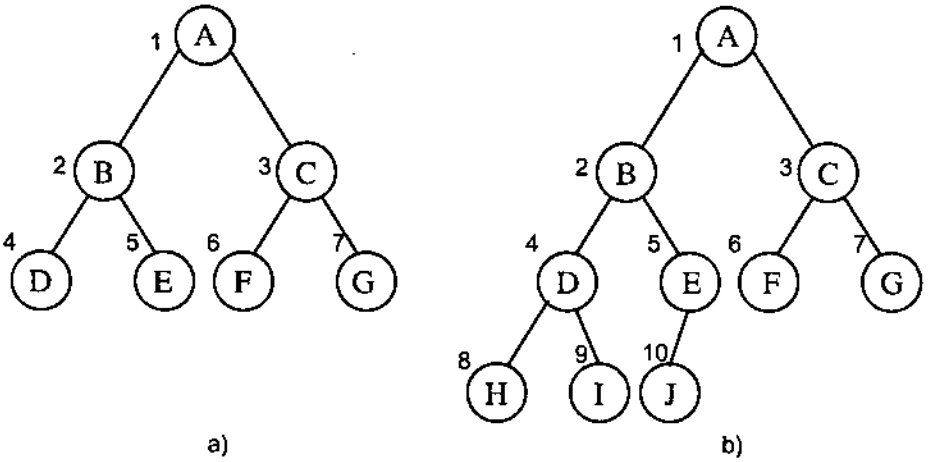
Ví dụ : x = 3,25 thì $\lceil x \rceil = 4$

x = 3 $\lceil x \rceil = 3$)

5.3. BIỂU DIỄN TRONG MÁY CỦA CÂY NHỊ PHÂN

1. Lưu trữ kế tiếp

Ta hãy xét một cây nhị phân đầy đủ (hoặc cây nhị phân hoàn chỉnh). Ta thấy : với dạng cây này thì ta có thể đánh số liên tiếp cac nút trên cây theo trình tự từ mức 1, hết mức này đến mức khác và từ trái sang phải đối với các nút trên cùng một mức, như ở hình 5.11.



HÌNH 5.11

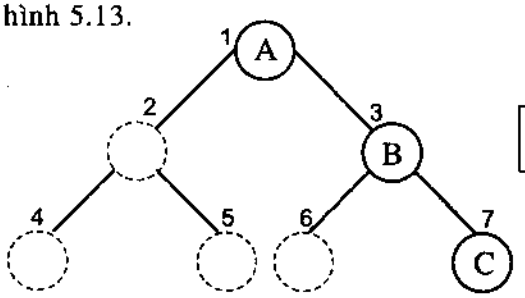
Với cách đánh số này thì nếu cha có số thứ tự là i thì rõ ràng con sẽ có số thứ tự là 2i và 2i + 1. Hoặc ngược lại : nếu con có số thứ tự là j thì cha sẽ có số thứ tự là $\lfloor j/2 \rfloor$ ($\lfloor x \rfloor$ chỉ số nguyên dưới của x. Ví dụ $\lfloor 3{,}25 \rfloor = 3$).

Như vậy ta có thể lưu trữ cây nhị phân bằng một vectơ lưu trữ V, theo nguyên tắc là nút thứ i trên cây sẽ được lưu trữ ở V[i]. Và với cách lưu trữ này thì biết phần tử chứa nút cha sẽ suy ra ngay phần tử chứa nút con và ngược lại. Như vậy cây ở hình 5.11b) thì vectơ lưu trữ V sẽ gồm 10 phần tử và có thể minh họa như sau :

A	B	C	D	E	F	G	H	I	J
V[1]	V[2]	V[3]	V[4]	V[5]	V[6]	V[7]	V[8]	V[9]	V[10]

HÌNH 5.12

Ta cũng thấy ngay: cách lưu trữ này chỉ thích hợp với cây nhị phân đầy đủ hoặc hoàn chỉnh. Còn đối với cây nhị phân dạng bất kì thì theo quy tắc đánh số đã nêu ở trên (để đảm bảo luật tính chỉ số (địa chỉ) như đã nêu) có thể sẽ xuất hiện khá nhiều phần tử lưu trữ của V phải để trống, nghĩa là sẽ gây ra lãng phí bộ nhớ ! Ví dụ với cây dưới đây thì vectơ lưu trữ có dạng như trong hình 5.13.



V[1]	V[2]	V[3]	V[4]	V[5]	V[6]	V[7]
A	∅	B	∅	∅	∅	C

(kí hiệu ∅ nghĩa là để trống)

HÌNH 5.13

2. Lưu trữ móc nối

Một cách lưu trữ thích hợp cho mọi dạng cây nhị phân, đó là lưu trữ móc nối.

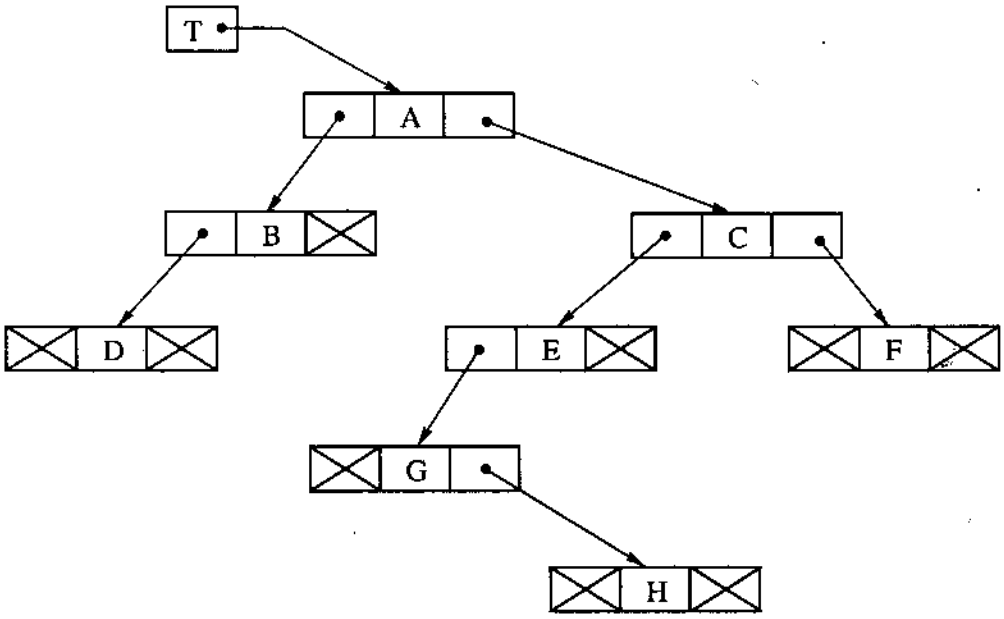
Ở đây, mỗi nút trên cây được lưu trữ bởi một phần tử có quy cách như sau :

Trường INFO chứa các thông tin (dữ liệu) của nút

Trường LPTR chứa địa chỉ của nút gốc gây con trái (trỏ tới cây con trái).

Trường RPTR chứa địa chỉ của nút gốc cây con phải (trỏ tới cây con phải).

Ví dụ : với cây nhị phân ở hình 5.5 thì dạng lưu trữ móc nối sẽ như hình 5.14 :



HÌNH 5.14

Tất nhiên để có thể truy cập vào các nút trên cây ta phải biết được con trỏ T, trỏ tới nút gốc cây (địa chỉ nút gốc).

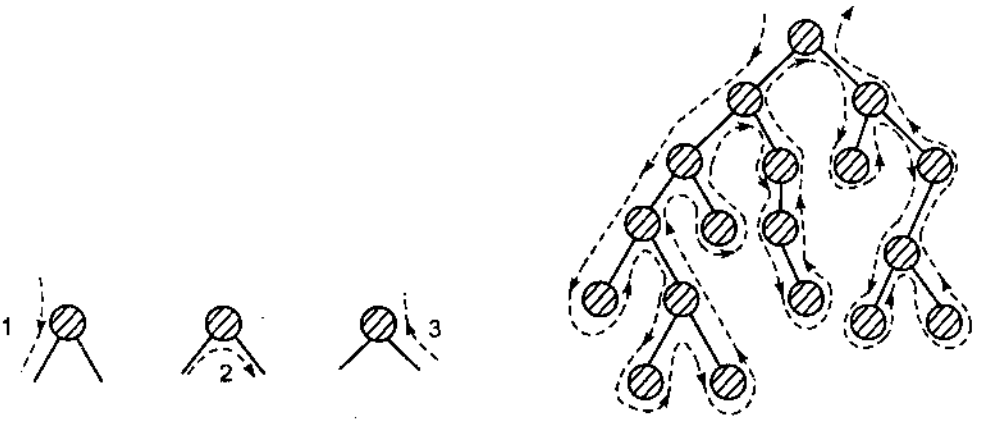
Người ta quy ước : nếu cây rỗng thì T = null. Rõ ràng với cách lưu trữ này số phần tử lưu trữ, ứng với số nút có trên cây (dù cây có dạng thế nào cũng vậy). Nhưng việc truy cập vào các nút thì chỉ có thể thực hiện được từ nút cha, nghĩa là biết nút cha thì biết được địa chỉ nút con, còn ngược lại thì không thể được.

5.4. PHÉP DUYỆT CÂY NHỊ PHÂN (TRAVERSING BINARY TREE)

Có rất nhiều ứng dụng, trong đó đòi hỏi ta phải xử lí lần lượt các nút trên cây nhị phân theo một thứ tự nào đó, nghĩa là "thăm" lần lượt các nút đó sao cho mỗi nút chỉ được thăm một lần thôi. Ta gọi đó là *phép duyệt* một cây.

Xét một cây nhị phân như hình 5.15.

Nếu ta dạo quanh cây theo chiều mũi tên như hình vẽ thì ta sẽ thấy đối với mỗi nút ta sẽ gặp 3 lần.



HÌNH 5.15

Vì vậy người ta có thể nêu lên 3 phép duyệt.

Các phép đó được định nghĩa đệ quy như sau :

1) Duyệt theo thứ tự trước (preorder traversal)

a) Nếu cây rỗng thì không làm gì

b) Nếu cây không rỗng thì :

– Thăm gốc
– Duyệt cây con trái theo thứ tự trước
– Duyệt cây con phải theo thứ tự trước

2) Duyệt theo thứ tự giữa (inorder traversal)

a) Nếu cây rỗng thì không làm gì

b) Nếu cây không rỗng thì :

– Duyệt cây con trái theo thứ tự giữa
– Thăm gốc
– Duyệt cây con phải theo thứ tự giữa

3) Duyệt theo thứ tự sau (postorder traversal)

a) Nếu cây rỗng thì không làm gì

b) Nếu cây không rỗng thì

– Duyệt cây con trái theo thứ tự sau

– Duyệt cây con phải theo thứ tự sau

– Thăm gốc

Dựa vào định nghĩa này ta có thể xác định được thứ tự các nút được thăm theo mỗi cách duyệt nêu trên.

Chẳng hạn với cây nhị phân ở hình 5.5 thì thứ tự được "thăm" của các nút sẽ như sau :

a) Duyệt theo thứ tự trước

A B D C E G H F

b) Duyệt theo thứ tự giữa

D B A G H E C F

c) Duyệt theo thứ tự sau :

D B H G E F C A

Còn với cây biểu diễn biểu thức số học như ở hình 5.9 thì kết quả sẽ là :

a) + – x * 2 y ↑ / y z 3

b) x – 2 * y + y / z ↑ 3

c) x 2 y * – y z / 3 ↑ +

Ta thấy với cây biểu diễn biểu thức số học thì

– Phép duyệt theo thứ tự trước sẽ cho ta dạng tiền tố của biểu thức.

– Phép duyệt theo thứ tự sau sẽ cho ta dạng hậu tố của biểu thức.

– Phép duyệt theo thứ tự giữa sẽ cho ta dạng trung tố, nhưng thiếu các dấu ngoặc cần thiết (nên dạng này không có giá trị sử dụng).

Rõ ràng nhận xét này cũng giúp ta xác định được dạng tiền tố, hoặc hậu tố của biểu thức khi ta dựng được cây nhị phân biểu diễn nó.

• Với các phép duyệt được định nghĩa đệ quy như trên, việc viết các thủ tục đệ quy tương ứng sẽ khá dễ dàng. Chẳng hạn thủ tục đệ quy của phép duyệt cây nhị phân theo thứ tự trước có thể diễn đạt bởi :

Procedure RPREORDER (T) ;

{ở đây T là con trỏ, trỏ tới nút gốc cây nhị phân được cài đặt trong máy dưới dạng móc nối như đã nêu ở 5.3. Phép "Thăm" ở đây được thể hiện bởi phép in ra nội dung của trường INFO}

```
1. if   T ≠ null   then  begin
                         write (INFO (T)) ;
2.                       call RPREORDER (LPTR (T)) ;
3.                       call RPREORDER (RPTR (T))
                              end ;
4. return
```

Với phép duyệt theo thứ tự giữa, thứ tự sau giải thuật đệ quy cũng được thể hiện tương tự (người đọc hãy tự viết).

Việc xây dựng một thủ tục không đệ quy để thực hiện các phép duyệt sẽ gặp một khó khăn chính là ở chỗ từ nút cha nếu đã truy cập vào nút con rồi thì từ nút con ta sẽ không tìm được địa chỉ của nút cha nữa. Chẳng hạn, với phép duyệt theo thứ tự trước : khi "thăm" một nút rồi ta sẽ phải xuống nút con trái của nó để thực hiện duyệt cây con trái. Nếu ta truy cập ngay vào nút con trái thì sau đó không thể tìm được địa chỉ nút con phải của cha nó nữa (vì sau đó còn phải duyệt cây con phải này). Cho nên trước khi truy cập vào nút con trái thì phải bảo lưu địa chỉ của nút con phải. Đối với cây con cũng phải làm tương tự. Ta thấy ngay : địa chỉ nút con phải được bảo lưu trước, khi "đi xuống" cây con trái, sẽ được khôi phục sau khi "đi lên" (để chuyển sang cây con phải). Ở đây sẽ xuất hiện cơ chế "vào – sau – ra – trước". Vì vậy trong giải thuật "phi đệ quy", người ta phải dùng tới stack. Và dĩ nhiên, giải thuật này sẽ phức tạp hơn nhiều nên ta sẽ không đi sâu vào.

5.5. BIỂU DIỄN CÂY TỔNG QUÁT BẰNG CÂY NHỊ PHÂN

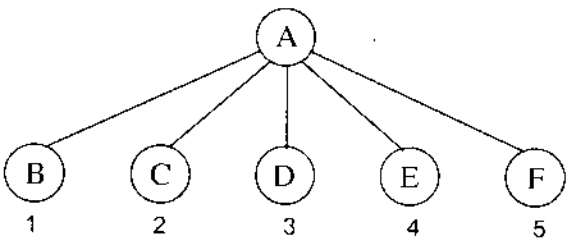
Một cách tự nhiên, ta có thể nghĩ ngay tới cách biểu diễn cây tổng quát, theo kiểu móc nối như đã làm đối với cây nhị phân, nghĩa là ở mỗi nút đều có chứa các con trỏ, trỏ tới cây con của nút đó. Như vậy : với cây cấp m (số con tối đa của mỗi nút là m) thì quy cách mỗi nút : ngoài trường INFO chứa thông tin ứng với nút đó, còn chứa m trường mối nối.

Rõ ràng là nếu trên cây mà các nút thường ít con (so với m) thì số "mối nối không" sẽ chiếm quá nhiều và điều đó sẽ gây ra lãng phí bộ nhớ.

Một trong các cách khác, khắc phục được nhược điểm này, là biểu diễn cây tổng quát bằng một cây nhị phân, mà được gọi là *cây nhị phân tương đương*.

Đối với một cây tổng quát ta có thể gán số thứ tự cho các cây con của mỗi nút. Giả sử trên hình vẽ ta đánh số thứ tự từ trái sang phải.

Chẳng hạn với nút có 5 con sau đây, thì sẽ đánh số như



HÌNH 5.16

Và lúc đó ta gọi : nút 1 là "con cả" của nút A, nút 2 là "em sát nút 1", nút 3 là "em sát nút 2" v.v..

Ta thấy : có một quan hệ 1 – 1 giữa nút cha và nút "con cả", giữa một nút và nút "em sát nó". Vì vậy đối với cây tổng quát : với mỗi nút người ta chỉ chú ý tới hai quan hệ này là đủ. Từ đó quy cách của mỗi nút trên cây nhị phân tương đương, có dạng :

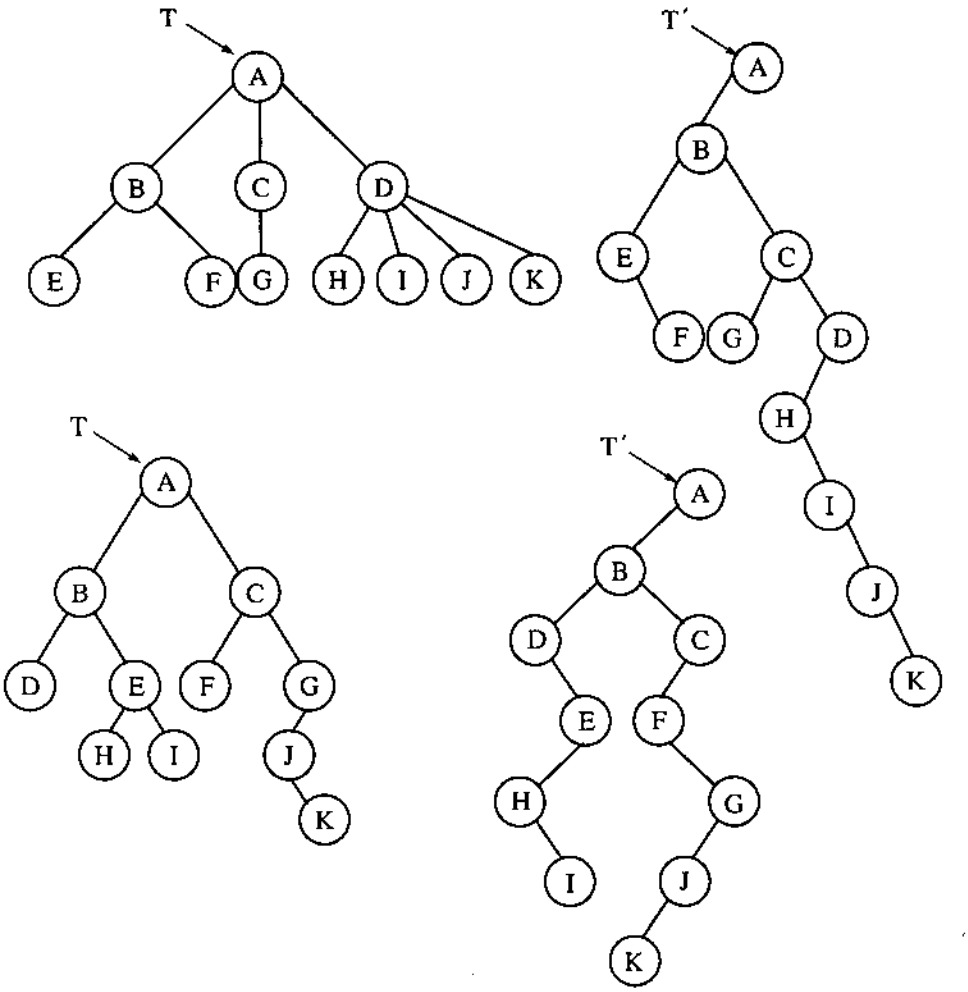
LCHD	INFO	RSIB

với LCHD là trường con trỏ, ở đó chứa địa chỉ của nút "con cả"

RSIB là trường con trỏ, ở đó chứa địa chỉ của nút "em sát nó"

Như vậy với bất kì một cây nào cũng có một và chỉ một cây nhị phân tương đương với nó. Điều đó cũng có nghĩa là : với một cây tổng quát cho, thì biểu diễn trong máy của nó là cây nhị phân tương đương. Các phép xử lí trên cây tổng quát đều thực hiện qua cây nhị phân tương đương này và kết quả sau khi chuyển đổi lại, sẽ phải khớp với ý đồ xử lí đối với cây tổng quát.

Sau đây là ví dụ minh họa một vài cây nhị phân tương đương ứng với cây tổng quát cho :



T là con trỏ, trỏ tới gốc cây tổng quát

T' là con trỏ, trỏ tới gốc cây nhị phân tương đương của cây T)

HÌNH 5.17

Ta thấy :
- Gốc của cây nhị phân tương đương T' không có con phải.
- Cây nhị phân tương đương T' của một cây nhị phân T thường khác nó.

5.6. ÁP DỤNG

5.6.1. Cấu trúc cây trong sắp xếp

Như ta đã biết, sắp xếp "NHANH" hay sắp xếp kiểu phân đoạn là một cải tiến khá tốt. Thời gian thực hiện trung bình của giải thuật này đã đạt được là : $T_{tb}(n) = O(n \log_2 n)$.

Tuy nhiên trường hợp xấu thì thời gian thực hiện vẫn là

$$T_x(n) = O(n^2)$$

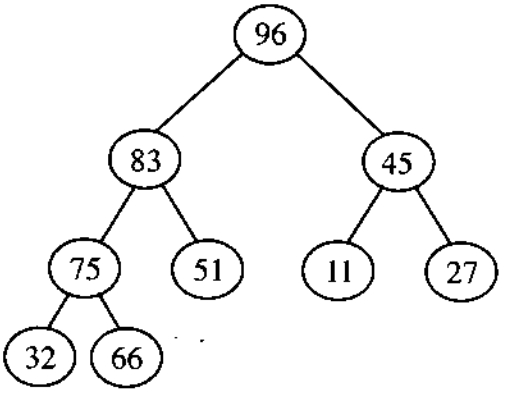
Phương pháp sắp xếp sau đây, dựa trên cấu trúc dữ liệu là cấu trúc cây, sẽ khắc phục được nhược điểm của sắp xếp "NHANH" ; cụ thể là $T_x(n) = T_{tb}(n) = = O(n \log_2 n)$

Cấu trúc cây ở đây được thể hiện dưới một dạng đặc biệt, gọi là *đống* (heap).

1. Cấu trúc : "đống"

Đống là một cây nhị phân hoàn chỉnh mà mỗi nút được gắn với một số, sao cho số ở nút cha bao giờ cũng lớn hơn số ở nút con.

Ví dụ : Cây trên hình 5.18 là một "đống" với 9 nút.



HÌNH 5.18

Do cây nhị phân hoàn chỉnh có thể lưu trữ rất thuận lợi theo kiểu kế tiếp, nên đống luôn được biểu diễn trong máy dưới dạng một vectơ. Với đống ở ví dụ trên thì biểu diễn trong máy của nó như sau :

96	83	45	75	51	11	27	32	66
1	2	3	4	5	6	7	8	9

HÌNH 5.19

Ta cũng thấy thêm rằng : số ứng với gốc của đống chính là số lớn nhất so với mọi số khác trong đống.

2. Phép tạo đống

Xét một cây nhị phân hoàn chỉnh mà ứng với mỗi nút đã gắn vào đó một số (coi như đó là nội dung của mỗi nút). Rõ ràng có thể biểu diễn cây này trong máy bởi một vectơ mà các phần tử là các số ứng với các nút. Vậy thì ngược lại, nếu có một dãy số trong máy, biểu diễn dưới dạng một vectơ, ta cũng có thể coi đó là vectơ biểu diễn cho một cây nhị phân hoàn chỉnh, với mỗi số ứng với mỗi nút, như đã nêu trên. Thông thường cây này chưa phải là đống.

Ví dụ : với dãy số sau, được biểu diễn trong máy dưới dạng một vectơ, thì có thể coi như nó là biểu diễn của một cây nhị phân hoàn chỉnh theo hình 5.20 :

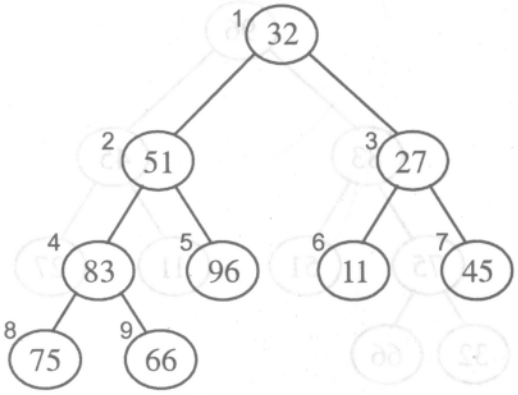
Dãy số A cho :

32 51 27 83 96 11 45 75 66

Vectơ biểu diễn A trong máy :

1	2	3	4	5	6	7	8	9
32	51	27	83	96	11	45	75	66

Cây nhị phân hoàn chỉnh tương ứng :



HÌNH 5.20

Vấn đề đặt ra bây giờ là :

Với một cây nhị phân hoàn chỉnh như trên, muốn tạo nó thành đống thì làm thế nào ?

Trước hết ta có nhận xét :

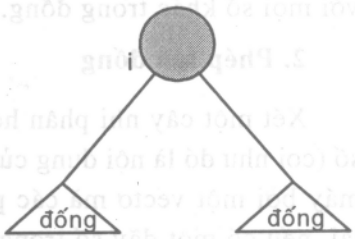
– Nếu một cây nhị phân hoàn chỉnh đã là đống thì các cây con trên nó cũng đều là đống.

– Trên cây nhị phân hoàn chỉnh có n nút thì chỉ có $\lfloor n/2 \rfloor$ nút được gọi là "cha" thôi.

– Nút lá bao giờ cũng có thể coi là đống được.

Từ đó có thể thực hiện phép tạo đống bằng cách : tiến hành theo kiểu "từ đáy lên" (bottom up). Do lá vốn là đống rồi nên ta tạo thành đống dần cho các cây con mà gốc của nó có số thứ tự từ $\lfloor n/2 \rfloor$ trở xuống. Như với cây trên thì xử lí các cây con với gốc lần lượt có số thứ tự là 4, 3, 2, 1. Việc xử lí với các cây này đều tương tự như nhau, đó chính là việc giải quyết bài toán :

"Hãy tạo thành đống cho một cây nhị phân hoàn chỉnh (với các số ứng với các nút) mà gốc cây này có thứ tự là i (theo cách đánh số thứ tự khi lưu trữ kế tiếp đối với cây nhị phân hoàn chỉnh) và 2 con của nút gốc đã là đống rồi.



HÌNH 5.21

Thủ tục sau đây sẽ thể hiện giải thuật thực hiện phép xử lí này.

Procedure ADJUST (i,n) ;

{ở đây i là thứ tự của nút gốc cây con cần xét, thuộc cây nhị phân hoàn chỉnh có n nút (đã được đánh số thứ tự theo quy tắc như đã nêu khi lưu trữ kế tiếp) và được lưu trữ trong máy bởi vectơ A}

1. {Bảo lưu số ở nút i và ghi nhận chỉ số của nút con trái}

KEY := A[i] ;

j := 2 * i ;

2. **while** j ≤ n **do begin**

3. **if** j < n **and** A[j] < A[j + 1] **then** j :=j + 1

{nếu số ở con phải lớn hơn, tạm gọi : "nếu con lớn là con phải", thì ghi nhận chỉ số của con phải, nghĩa là tìm con lớn nhất trong 2 con}

4. **if** KEY > A[j] **then begin**

A[⌊j/2⌋] := KEY ;

return

end ;

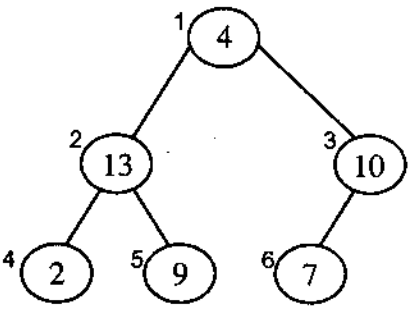
5. A[⌊j/2⌋] := A[j] ; {đưa số lớn hơn lên vị trí "cha"

j := 2 * j ; {tiếp tục xuống "con trái"}

end

6. A[⌊j/2⌋] := KEY ;

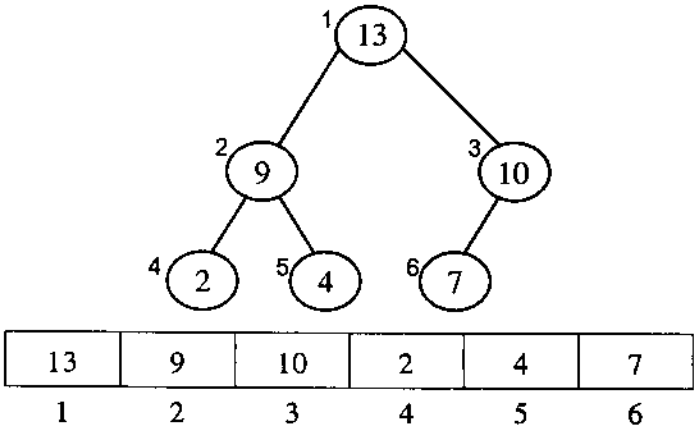
7. **return**.



HÌNH 5.22

Ví dụ : với cây hình 5.22 có 2 cây con đã là đống rồi :

Nếu áp dụng giải thuật trên, cuối cùng ta sẽ có đống như hình 5.23.



HÌNH 5.23

Với cây nhị phân hoàn chỉnh, có n nút biểu diễn bởi vectơ A trong máy, thì theo nhận xét đã nêu ở trên, câu lệnh sau đây sẽ thực hiện việc tạo cây đó thành đống.

for i := ⌊j/2⌋ **down to** 1 **do call** ADJUST (i, n)

3. Sắp xếp kiểu "vun đống" (heap – sort)

Sắp xếp kiểu vun đống cũng thực hiện các phép xử lí cơ bản như sắp xếp kiểu lựa chọn. Cụ thể là :

"Số lớn nhất trong dãy số đang xét sẽ được chọn ra và sau đó đổi chỗ với số ở cuối dãy".

Việc chọn số lớn nhất ở đây đã được cải tiến nhờ cấu trúc đống, vì nếu có đống thì số lớn nhất đang nằm ngay ở gốc (mà ta sẽ gọi là "đỉnh đống") nó ứng với phần tử đầu tiên của vectơ biểu diễn đống.

Xét một dãy số cho, cần được sắp xếp theo thứ tự tăng dần. Ta có thể coi dãy này như vectơ biểu diễn cho một cây nhị phân hoàn chỉnh, thường cây này chưa phải là đống.

Người ta áp dụng giải thuật ADJUST để tạo nó thành đống. Số đang nằm ở đỉnh đống chính là số lớn nhất. Nó sẽ được đổi chỗ với số đang ở cuối đống, vì khi sắp xếp theo thứ tự tăng dần số lớn nhất phải nằm ở vị trí cuối. Lúc này đã có 1 số vào đúng vị trí sắp xếp của nó. Dãy số còn lại (sau khi đã bớt đi số cuối) lại được coi như ứng với một cây nhị phân hoàn chỉnh. Bây giờ lại tạo nó thành đống (ta sẽ gọi là "vun đống") và lặp lại công việc như đã nêu. Cuối cùng khi đống chỉ còn một nút, ứng với một số, thì công việc sắp xếp đã hoàn thành.

Như vậy, sắp xếp kiểu vun đống bao gồm hai giai đoạn chính :

1. Từ dãy số cho, quan niệm như một vectơ biểu diễn cho một cây nhị phân hoàn chỉnh, tạo nên một đống đầu tiên (Ta gọi là giai đoạn tạo đống ban đầu).

2. Đổi chỗ giữa số ở đỉnh đống với số ở cuối đống sau đó vun đống mới.

Quá trình này sẽ được lặp lại cho tới khi đống chỉ còn một nút thôi. (Ta gọi là giai đoạn sắp xếp).

Giải thuật sắp xếp kiểu vun đống được thể hiện như sau :

Procedure HEAP – SORT (A, n)

{A là vectơ có n phần tử, biểu diễn dãy số cần sắp xếp}

1. {Tạo đống ban đầu}

for i := ⌊n/2⌋ **down to** 1 **do call** ADJUST (i, n) ;

2. {Sắp xếp}

for i := n – 1 **down to** 1 **do begin**

A[1] ↔ A[i + 1] ;

call ADJUST (1, i)

end ;

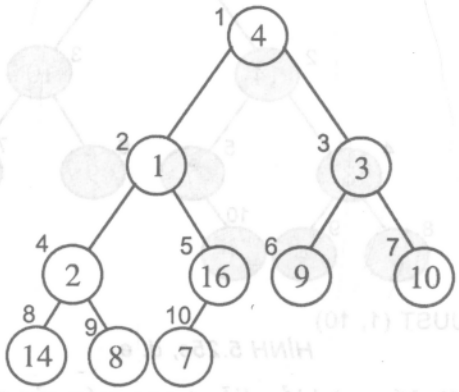
3. **return**

Sau đây là một ví dụ minh họa

Dãy số A cần sắp xếp :

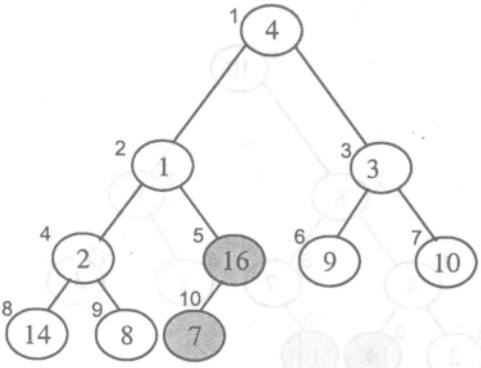
	1	2	3	4	5	6	7	8	9	10
A	4	1	3	2	16	9	10	14	8	7

Cây nhị phân hoàn chỉnh ứng với A

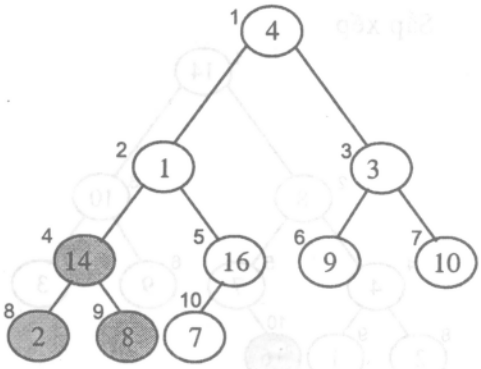


HÌNH 5.24

Tạo đống ban đầu

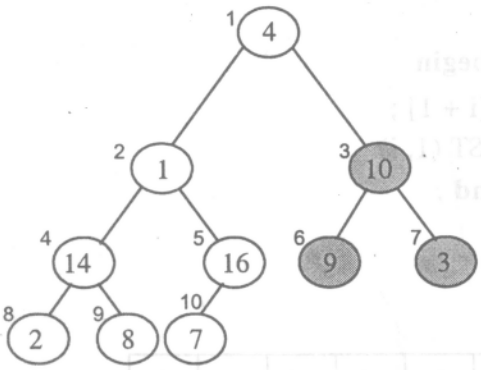


a) Thực hiện ADJUST (5, 10)

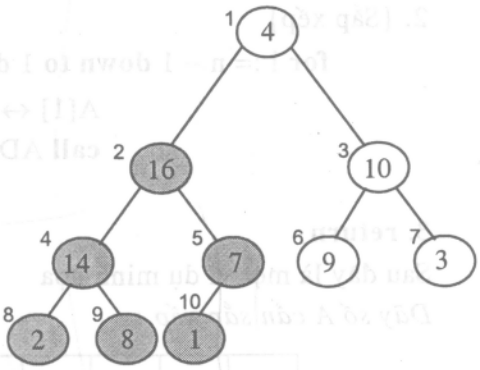


b) Thực hiện ADJUST (4, 10)

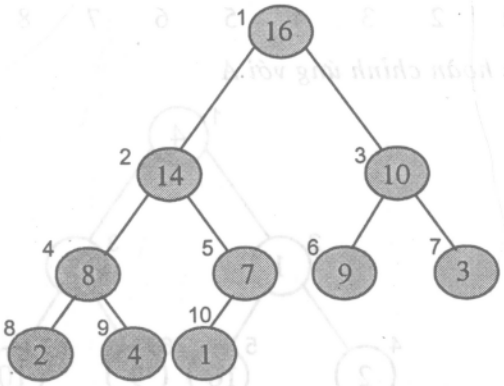
HÌNH 5.25a,b



c) Thực hiện ADJUST (3, 10)



d) Thực hiện ADJUST (2, 10)



e) Thực hiện ADJUST (1, 10)

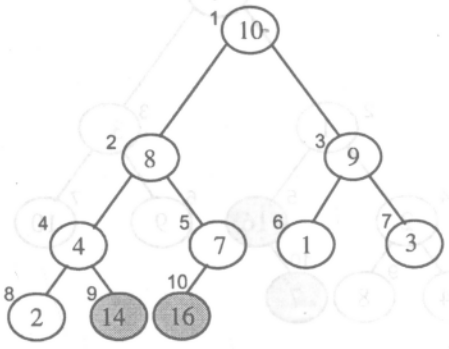
HÌNH 5.25c, d, e.

Lúc này cây đã là đống và biểu diễn trong máy của nó là vectơ A như sau :

A	16	14	10	8	7	9	3	2	4	1
	1	2	3	4	5	6	7	8	9	10

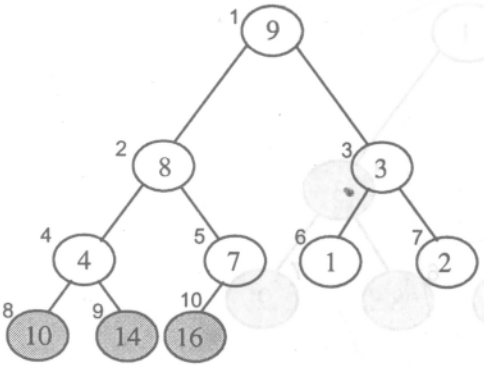
Sắp xếp

a) Đổi chỗ lần 1 : giữa A[1] và A[10], và vun thành đống cho cây với 9 nút còn lại số 16 đã vào đúng vị trí

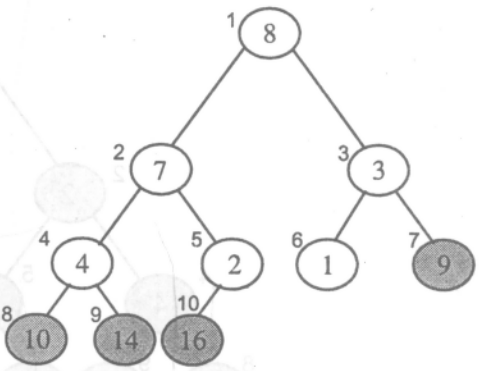


b) Đổi chỗ lần 2 và vun đống cho cây với 8 nút còn lại. Các số 14, 16 đã vào đúng vị trí

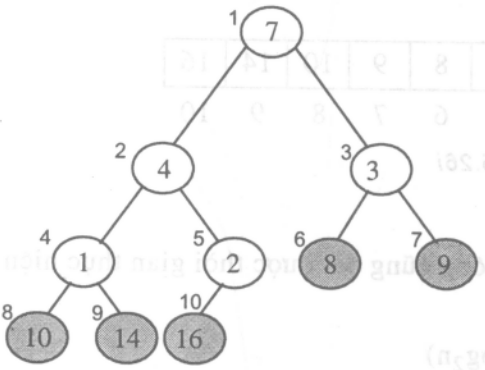
HÌNH 5.26a, b



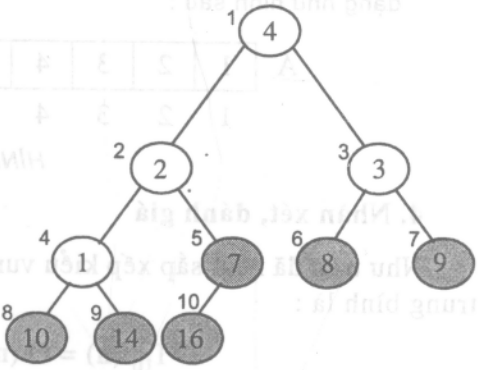
c) Đổi chỗ lần 3 và vun đống cho cây với 7 nút còn lại. Các số 10, 14, 16 đã vào đúng vị trí



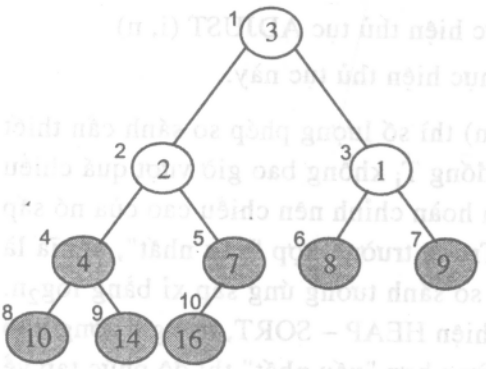
d) Đổi chỗ lần 4 và vun đống cho cây với 6 nút còn lại. Các số 9, 10, 14, 16 đã vào đúng vị trí



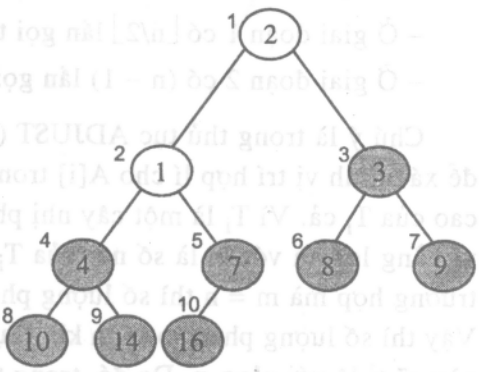
e) Đổi chỗ lần 5, và vun đống cho cây với 5 nút còn lại. Các số 8, 9, 10, 14, 16 đã vào đúng vị trí



f) Đổi chỗ lần 6 và vun đống cho cây với 4 nút còn lại. Các số 7, 8, 9, 10, 14, 16 đã vào đúng vị trí.

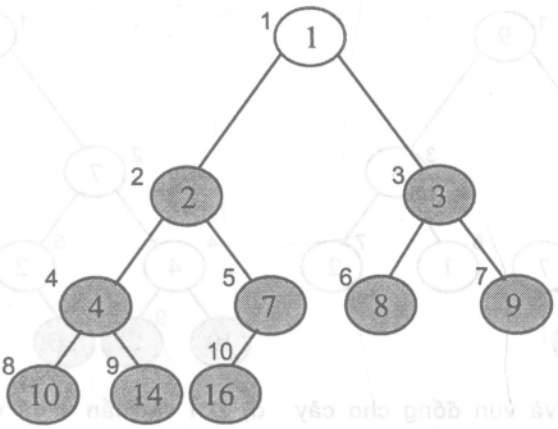


g) Đổi chỗ lần 7 và vun đống cho cây với 3 nút còn lại. Các số 4, 7, 8, 9, 10, 14, 16 đã vào đúng vị trí.



h) Đổi chỗ lần 8 và vun đống cho cây với 2 nút còn lại. Các số 3, 4, 7, 8, 9, 10, 14, 16 đã vào đúng vị trí

HÌNH 5.26c, d, e, f, g, h



i) Đổi chỗ lần 9 và bây giờ đống chỉ còn một nút. Dãy số A đã được sắp xếp và có dạng như hình sau :

A	1	2	3	4	7	8	9	10	14	16
	1	2	3	4	5	6	7	8	9	10

HÌNH 5.26i

4. Nhận xét, đánh giá

Như trên đã nêu, sắp xếp kiểu vun đống cũng đạt được thời gian thực hiện trung bình là :

$$T_{tb}(n) = O(n \log_2 n)$$

Nay ta chỉ xét trường hợp "xấu nhất".

Theo giải thuật HEAP – SORT ở trên :

– Ở giai đoạn 1 có $\lfloor n/2 \rfloor$ lần gọi thực hiện thủ tục ADJUST (i, n)

– Ở giai đoạn 2 có (n – 1) lần gọi thực hiện thủ tục này.

Chú ý là trong thủ tục ADJUST (i, n) thì số lượng phép so sánh cần thiết để xác định vị trí hợp lí cho A[i] trong đống T_i không bao giờ vượt quá chiều cao của T_i cả. Vì T_i là một cây nhị phân hoàn chỉnh nên chiều cao của nó sấp xỉ bằng $\log_2 m$ với m là số nút của T_i. Trong trường hợp "xấu nhất", nghĩa là trường hợp mà m = n thì số lượng phép so sánh tương ứng sấp xỉ bằng $\log_2 n$. Vậy thì số lượng phép so sánh khi thực hiện HEAP – SORT, trong trường hợp này sẽ tỉ lệ với $n\log_2 n$. Do đó, trong trường hợp "xấu nhất" thì độ phức tạp về thời gian của HEAP – SORT sẽ là : $T_x(n) = O(n \log_2 n)$.

Như vậy, rõ ràng HEAP – SORT có ưu điểm hơn QUICK – SORT về khía cạnh này !

5.6.2. Cấu trúc cây trong tìm kiếm

Ta đã biết rằng với một dãy số cho, đã được sắp thứ tự, chẳng hạn theo thứ tự tăng dần và được lưu trữ kế tiếp trong máy bởi một vectơ thì phép tìm kiếm nhị phân sẽ cho ta kết quả với thời gian thực hiện trung bình nhanh nhất :

$$T_{tb} = O(\log_2 n).$$

Nhưng nếu xét một dãy số với số lượng các phần tử luôn biến động, nghĩa là một danh sách mà phép bổ sung phần tử mới hoặc loại bỏ phần tử cũ thường xảy ra, thì cách lưu trữ kế tiếp cũng như giải thuật tìm kiếm nhị phân sẽ không còn thích hợp nữa.

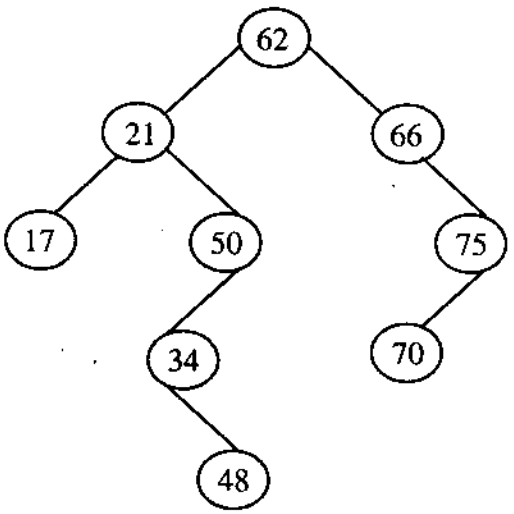
Trong trường hợp này, phương pháp tìm kiếm với cấu trúc dữ liệu được tổ chức dưới dạng cây, mà được gọi là *cây nhị phân tìm kiếm* (binary – search tree), và được lưu trữ theo kiểu móc nối, sẽ tỏ ra hữu hiệu hơn.

1. Cây nhị phân tìm kiếm

Cây nhị phân tìm kiếm ứng với một dãy số cho a_1, a_2, ..., a_n là một cây nhị phân mà ứng với mỗi nút người ta gắn vào đó một số thuộc dãy, sao cho đối với mọi nút trên cây tính chất sau luôn được thỏa mãn :

– Mọi số thuộc cây con trái của nút đó đều nhỏ hơn số ứng với nút đó.

– Mọi số thuộc cây con phải của nút đó đều lớn hơn số ứng với nút đó.



HÌNH 5.27

Cây trên hình 5.27 là một ví dụ.

2. Tìm kiếm và bổ sung trên "cây nhị phân tìm kiếm"

Xét một cây nhị phân tìm kiếm T (T là con trỏ trỏ tới gốc cây này) và cho một số X.

Nếu muốn tìm xem trong các số ứng với các nút trên cây có số nào bằng X không thì ta có thể thực hiện :

So sánh X với số ở gốc và một trong 4 tình huống sau sẽ xảy ra :

1) Cây rỗng (T = null) : X không có trên cây, phép tìm kiếm không thỏa.

2) X nhỏ hơn số ở gốc : Tìm kiếm được thực hiện tương tự với cây con trái của gốc.

3) X lớn hơn số ở gốc : Tìm kiếm được thực hiện tương tự với cây con phải của gốc.

4) X trùng với số ở gốc : phép tìm kiếm đã thỏa

Với cây nhị phân tìm kiếm như đã nêu ở trên

a) Nếu X = 34, ta sẽ thực hiện

– So sánh X với 62 : X < 62, ta chuyển sang cây con trái

– So sánh X với 21 : X > 21, ta chuyển sang cây con phải

– So sánh X với 50 : X < 50, ta chuyển sang cây con trái

– So sánh X với 34 : X = 34, phép tìm kiếm đã thỏa

b) Nếu X = 68, ta sẽ thực hiện

– So sánh X với 62 : X > 62, ta chuyển sang cây con phải

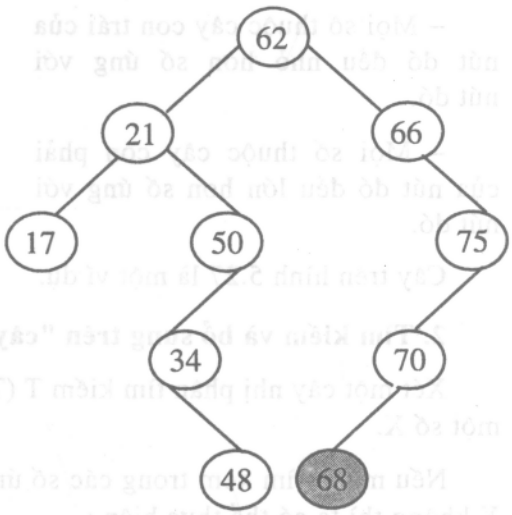
– So sánh X với 66 : X > 66, ta chuyển sang cây con phải

– So sánh X với 75 : X < 75, ta chuyển sang cây con trái

– So sánh X với 70 : X < 70, ta chuyển sang cây con trái nhưng cây này rỗng. Vậy phép tìm kiếm không thỏa !

Nếu muốn bổ sung một nút mới, ứng với một số X cho, thì trước hết ta vẫn phải tìm kiếm. Nếu phép tìm kiếm không thỏa, nghĩa là : X là một số mới, không có trên cây, thì nút mới ứng với X sẽ được bổ sung vào ngay vị trí của gốc cây rỗng. Phép bổ sung này đảm bảo cây đang xét vẫn là cây nhị phân tìm kiếm. Chẳng hạn với X = 68 như trên, sau phép bổ sung ta sẽ có cây nhị phân tìm kiếm như hình 5.28.

Ta cũng thấy ngay : phép bổ sung nút mới không làm xáo trộn gì cấu trúc cũ của cây cả !



HÌNH 5.28

Sau đây là giải thuật "Tìm kiếm, có bổ sung" :

Nếu phép tìm kiếm X không thỏa thì bổ sung nút mới, ứng với X, vào cây nhị phân tìm kiếm.

Ở đây cây nhị phân tìm kiếm được lưu trữ móc nối với quy cách mỗi nút như đã nêu ở 5.3 và trường INFO sẽ chứa số trong dãy số cho.

Procedure BST (T, X, q)

{Ở đây T là con trỏ, trỏ tới gốc cây nhị phân tìm kiếm. Nếu tìm kiếm đã thỏa thì q ghi nhận địa chỉ của nút chứa số bằng X, nếu tìm kiếm không thỏa thì q ghi nhận địa chỉ của nút mới được bổ sung vào}

1. {Khởi tạo con trỏ}

    p := null ; q : = T ;

2. {Tìm kiếm}

    while q ≠ null **do**

        case

        X < INFO (q) : p : = q ; q : = LPTR (q) ;

        X > INFO (q) : p : = q ; q : = RPTR (q) ;

        X = INFO (q) : **write** ('Đã thấy') ; **return** ;

        end case ;

3. {Bổ sung}

    call New (q) ; INFO (q) := X ; LPTR (q) := RPTR (q) := null ;

    case

    T = null : T := q ; {Trường hợp cây rỗng}

    X < INFO(p) : LPTR(p) := q ;

    X > INFO(p) : RPTR(p) := q ;

    end case

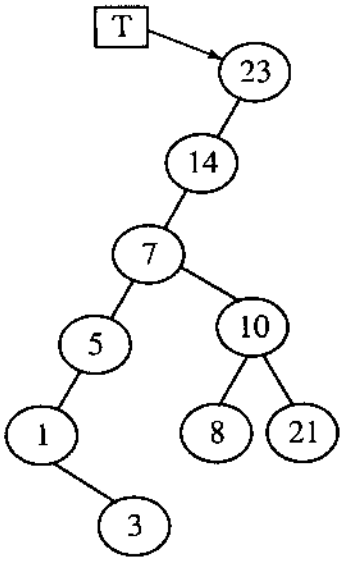
4. **Write** ('Đã bổ sung') ; **return**.

Chú ý rằng : giải thuật này cũng xử lí cả trường hợp T = null, nghĩa là khi cây nhị phân tìm kiếm còn rỗng. Điều đó sẽ cho phép ta dựng lên được cây nhị phân tìm kiếm ứng với một dãy số cho, xuất phát từ một cây rỗng.

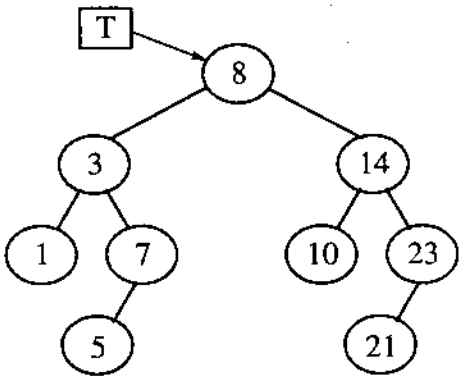
Ví dụ : với dãy số

23, 14, 7, 10, 8, 5, 1, 21, 3

nếu áp dụng giải thuật BST ta sẽ dựng được cây nhị phân tìm kiếm hình 5.29.



HÌNH 5.29



HÌNH 5.30

Ta cũng thấy : dạng của cây nhị phân tìm kiếm hoàn toàn phụ thuộc vào thứ tự của phần tử thuộc dãy cho. Nếu các phần tử của dãy trên được đưa vào theo thứ tự.

8, 3, 14, 1, 10, 23, 7, 21, 5

thì cây nhị phân tìm kiếm dựng được sẽ có dạng khác : đó là một cây nhị phân gần đầy, như hình 5.30.

3. Loại bỏ trên cây nhị phân tìm kiếm

Khi một nút, ứng với một số (được xác định qua phép tìm kiếm) bị loại ra khỏi cây nhị phân tìm kiếm thì giải quyết thế nào ?

Dĩ nhiên ta phải xử lí sao cho sau khi loại nút đó rồi, phần cây còn lại vẫn phải là một cây nhị phân tìm kiếm và việc này không gây nên sự xáo trộn dạng cây trước đó.

Vấn đề là phải tìm được nút "Thay thế" thích hợp để sửa lại con trỏ, trước đây đã trỏ tới nút bị loại này và các con trỏ ở nút thay thế.

Ta thấy có các tình huống sau :

Nếu nút bị loại là một nút lá thì không phải tìm nút thay thế nữa. Mối nối trước đây trỏ tới nó, sẽ được thay thế bởi null.

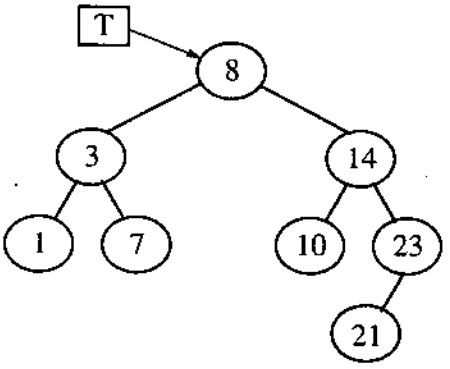
Nếu nút bị loại chỉ có một cây con thì nút thay thế nó chính là gốc của cây con này. Mối nối cũ trước đây trỏ tới nó, nay sẽ trỏ tới nút thay thế này.

Nếu nút bị loại có cả 2 con thì nút thay thế có thể hoặc là nút ứng với số nhỏ hơn ngay sát nó (nút cực phải của cây con trái nó), hoặc là nút ứng với số lớn hơn ngay sát nó (nút cực trái của cây con phải nó).

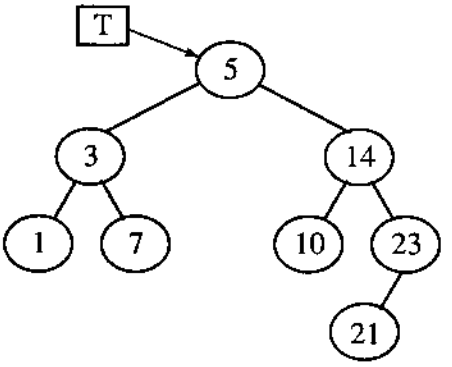
Như vậy ta sẽ phải sửa một số mối nối, rõ ràng việc sửa này không nhiều .

Ví dụ : với cây nhị phân tìm kiếm ở hình 5.30.

Nếu loại bỏ nút ứng với số 5 (nút lá) thì chỉ phải sửa một mối nối, và sau khi loại cây sẽ có dạng như hình 5.31.



HÌNH 5.31

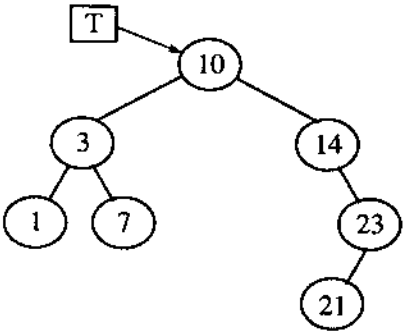


HÌNH 5.32

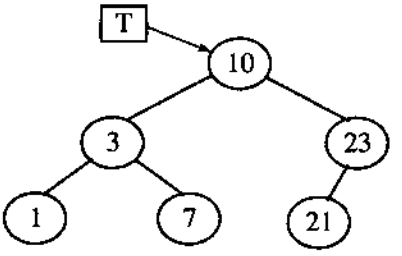
Nếu loại bỏ nút ứng với số 8, tức là nút có 2 con thì nút thay thế nó :

– Hoặc là nút ứng với số 5 (số nhỏ hơn 8 và sát ngay nó), khi đó ta phải sửa 4 mối nối và cây sẽ có dạng như hình 5.32).

– Hoặc là nút ứng với số 10 (số lớn hơn 8 và sát ngay nó), khi đó ta cũng phải sửa 4 mối nối và cây sẽ có dạng như hình 5.33.



HÌNH 5.33



HÌNH 5.34

Nếu với cây như trên, ta tiếp tục loại nút ứng với số 14 (nút chỉ có 1 con) thì nút thay thế nó chính là nút ứng với 23. Khi đó ta phải sửa một mối nối, và cây sẽ có dạng như hình 5.34.

4. Nhận xét đánh giá

Do khi bổ sung một nút mới ứng với số X, hoặc khi loại bỏ một nút có số bằng X trên cây nhị phân tìm kiếm, ta đều phải thực hiện phép tìm kiếm X. Việc sửa các mối nối khi bổ sung cũng như khi loại bỏ có chi phí thời gian không đáng kể, vì vậy việc đánh giá thời gian thực hiện phép bổ sung hay loại bỏ ở đây cũng dựa vào đánh giá thời gian thực hiện phép tìm kiếm. Tuy nhiên, thời gian này lại phụ thuộc và dạng cây nhị phân tìm kiếm.

Có thể thấy ngay trường hợp thuận lợi nhất thì chỉ cần một phép so sánh, nghĩa là X trùng ngay với số ở gốc cây : $T_t(n) = O(1)$ (n là số nút trên cây).

Còn trường hợp "xấu nhất" thì sao ?

Nếu cây nhị phân tìm kiếm là cây đầy đủ hoặc cây gần đầy (mà ta sẽ gọi là cây "cân đối") thì cùng lắm số phép so sánh cũng chỉ bằng chiều cao của cây, nghĩa là sấp xỉ $\log_2 n$.

Nhưng nếu cây nhị phân tìm kiếm có dạng cây suy biến, với chiều cao là n, thì số phép so sánh tối đa có thể bằng n. Trong trường hợp này $T_x(n) = O(n)$: phép tìm kiếm này đã không hơn gì tìm kiếm tuần tự.

May mắn là, người ta chứng minh được với phép tìm kiếm bằng cây nhị phân tìm kiếm thì $T_{tb}(n) = O(\log_2 n)$

Nhưng dù sao, trong quá trình xử lí với một dãy số biến động, dạng của cây nhị phân tìm kiếm rất có thể suy biến, với thời gian thực hiện tìm kiếm là O(n), thì đó cũng là mối lo ngại cho người dùng. Đó cũng chính là nhược điểm của cây nhị phân tìm kiếm và giải thuật BST nêu trên.

Việc nghiên cứu một dạng cây khác, với các phép xử lí đơn giản, để đảm bảo được tính "cân đối" của dạng cây và để độ phức tạp về thời gian thực hiện tìm kiếm luôn đạt được mức $O(\log_2 n)$ đã được đề ra và đã được giải quyết bởi khá nhiều tác giả, nhưng những kết quả đó sẽ không được nêu ra ở đây.

Câu hỏi và bài tập

5.1. Cho cây như hình 5.35.

Hãy trả lời các câu hỏi sau :

a) Nút nào là gốc ?

b) Nút nào là lá ?

c) Nút nào là nút nhánh ?

d) Các nút nào là anh em của I ?

e) Có bao nhiêu nút là tiền bối của J ?

f) Có bao nhiêu nút là hậu sinh của G ?

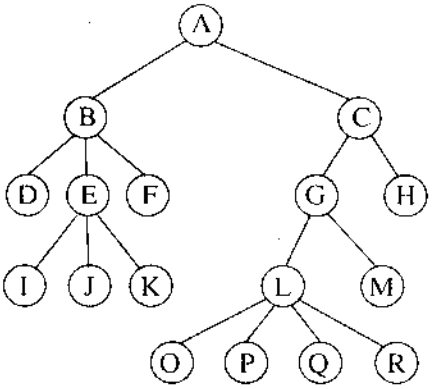
g) Mức của nút L là bao nhiêu ?

h) Cấp của B, của G là bao nhiêu ?

i) Cây này là cây cấp mấy ?

j) Chiều cao của cây này là bao nhiêu ?

k) Độ dài đường đi từ A tới J, từ A tới R là bao nhiêu ?



HÌNH 5.35

5.2. Hãy vẽ cây nhị phân có 7 nút

a) Với chiều cao lớn nhất.

b) Với chiều cao nhỏ nhất.

5.3. Vẽ cây nhị phân biểu diễn biểu thức số học sau :

a) (A * B + C) / D – E * F

b) ((A + B) * D) ↑ (E – F)

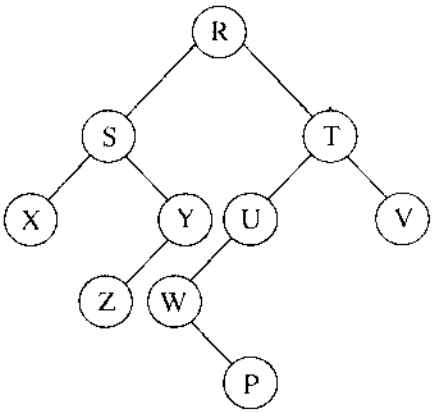
5.4. Cho cây nhị phân (hình 5.36)

Hãy viết dãy các nút được thăm khi duyệt cây này

a) Theo thứ tự trước.

b) Theo thứ tự giữa.

c) Theo thứ tự sau.



HÌNH 5.36

5.5. Cho biểu thức

$$E = (2x + y) * (5a - b)^3$$

a) Hãy vẽ cây nhị phân T biểu diễn E.

b) Muốn có biểu thức dạng tiền tố, dạng hậu tố thì phải duyệt cây T theo thứ tự nào. Nêu kết quả của các phép duyệt đó.

5.6. Cho cây nhị phân T, lưu trữ móc nối trong bộ nhớ. Hãy viết giải thuật đệ quy tính số nút của cây T.

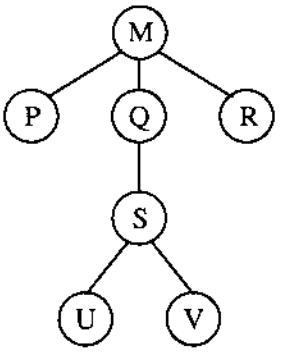
5.7. Một cây T có 9 nút. Phép duyệt cây đó theo thứ tự giữa và thứ tự trước cho dãy các nút được thăm như sau :

Thứ tự giữa : E A C K F H D B G

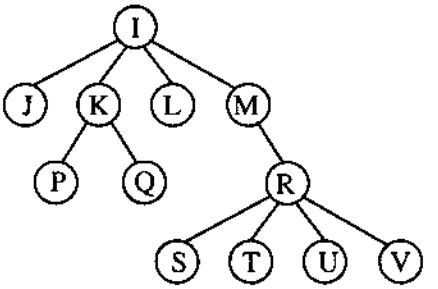
Thứ tự trước : F A E K C D H G B

Hãy chứng tỏ rằng ta có thể dựng được cây T đó và nêu kết quả.

5.8. Hãy dựng cây nhị phân tương đương với các cây hình 5.37, 5.38 :



HÌNH 5.37



HÌNH 5.38

5.9. Nếu xét cây nhị phân T, cùng với các cây tổng quát khác, thì T sẽ được biểu diễn bởi cây nhị phân tương đương T' ở trong máy.

a) Giả sử T có dạng như hình 5.39 thì T' có dạng như thế nào ?

b) Hãy so sánh dãy các nút được thăm ứng với T và T' lần lượt theo :

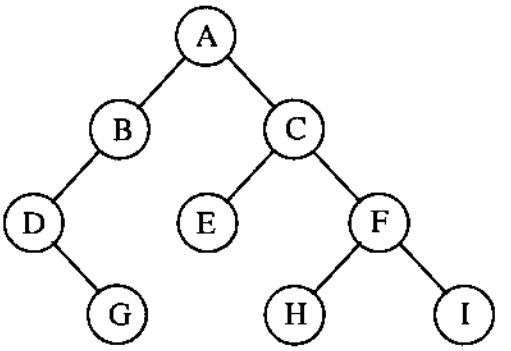
- Thứ tự trước.
- Thứ tự giữa.
- Thứ tự sau.

Anh (chị) có nhận xét gì không ?

c) Đối với cây tổng quát người ta chỉ nêu ra 2 phép duyệt :

- Duyệt theo thứ tự trước.
- Duyệt theo thứ tự sau.

Theo anh (chị) thì vì sao ?



HÌNH 5.39

5.10. Cấu trúc lưu trữ của "đống" và của "cây nhị phân tìm kiếm" là cấu trúc gì ?

Theo anh (chị) thì vì lí do gì người ta chọn cấu trúc lưu trữ đó.

5.11. Cho dãy số sau, được lưu trữ trong máy dưới dạng một vectơ

44, 30, 50, 22, 60, 55, 77, 65

a) Nếu coi như vectơ này biểu diễn một cây nhị phân hoàn chỉnh thì cây đó có dạng như thế nào ?

b) Cây này không phải là đống, Vì sao vậy ?

c) Có thể tạo thành đống cho cây này được. Muốn vậy phải làm thế nào ? Dựa vào giải thuật gì ? Minh họa dạng cây diễn biến trong quá trình này kèm theo dạng tương ứng của vectơ lưu trữ nó.

d) Nếu dựa vào đống đó để thực hiện sắp xếp thì làm thế nào ? Minh họa dạng cây qua các bước và thuyết minh cách làm.

5.12. Nếu sắp xếp theo thứ tự giảm dần mà vẫn thực hiện theo cách như của HEAP – SORT thì khái niệm "đống" có phải thay đổi không ? Thay đổi thế nào ?

5.13. Hãy dựa vào giải thuật BST để dựng lên cây nhị phân tìm kiếm ứng với dãy số cho sau đây :

33, 44, 50, 22, 60, 77, 35, 40, 10

5.14. Với dãy số như ở 5.13 thì :

a) Thứ tự các số trong dãy phải như thế nào để cây nhị phân tìm kiếm sẽ là một cây cao nhất.

b) Thứ tự các số trong dãy phải như thế nào để cây nhị phân tìm kiếm sẽ là một cây thấp nhất.

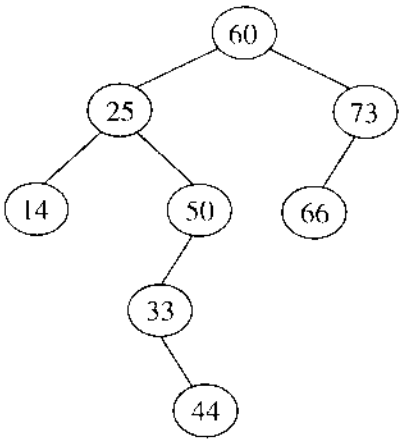
5.15. Cho cây nhị phân tìm kiếm (hình 5.40).

Hãy vẽ lại cây nhị phân tìm kiếm trong 3 trường hợp :

a) Sau khi loại nút 44

b) Sau khi loại nút 25

c) Sau khi loại nút 73



HÌNH 5.40

5.16. Cho cây nhị phân tìm kiếm như hình 5.40.

a) Hãy vẽ lại dạng cây sau khi lần lượt bổ sung các nút ứng với các số : 11, 20, 30 vào cây đó.

b) Với dạng cây mới này (sau khi đã bổ sung) nếu loại nút 25 đi thì dạng cây sẽ như thế nào ? Vẽ dạng cây đó.

Chương 6. ĐỒ THỊ (GRAPH)

6.1. ĐỊNH NGHĨA VÀ MỘT SỐ KHÁI NIỆM

Một đồ thị G(V, E) là một tập bao gồm 2 tập con :

– Tập hữu hạn V, không rỗng, của các phần tử mà ta gọi là *đỉnh* (vertices).

– Tập hữu hạn E, của các cặp đỉnh, mà mỗi cặp ta gọi là một *cung* (edge).

Bản đồ đường bộ giữa các thành phố trong một khu vực là một đồ thị với thành phố là đỉnh, đường nối giữa 2 thành phố là cung.

Mạng máy tính của một công ty, sơ đồ mạch điện của một khu chung cư, cấu trúc phân tử của các hợp chất hữu cơ v.v... tất cả đều có dạng đồ thị.

Nếu u và v là hai đỉnh trên đồ thị mà thứ tự được phân biệt trong từng cặp, nghĩa là (u, v) khác với (v, u) thì đồ thị được gọi là *đồ thị có hướng* (directed graph).

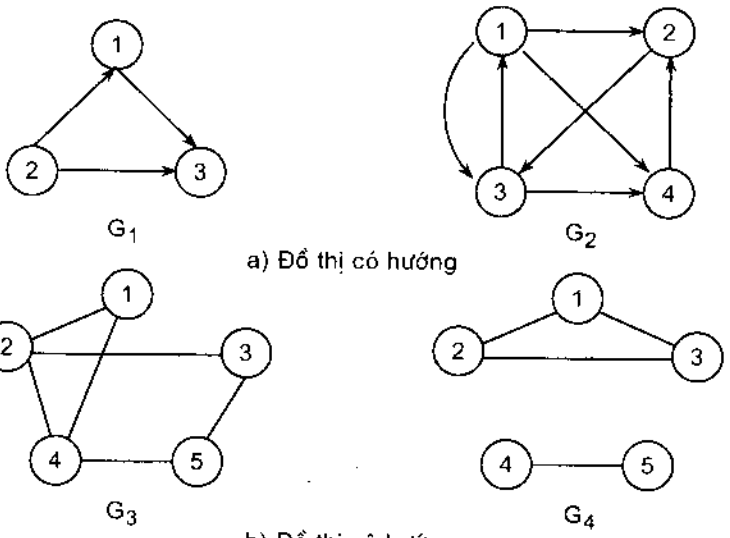
Lúc đó (u, v) được gọi là cung từ u tới v còn (v, u) được gọi là cung từ v tới u.

Nếu thứ tự hai đỉnh của một cung không được coi trọng thì đồ thị được gọi là *đồ thị vô hướng* (undirected graph) và (u, v) được gọi là cung giữa u và v hoặc cung giữa v và u.

Trên hình vẽ cung có hướng được biểu diễn bằng mũi tên còn cung vô hướng thì bằng một đoạn thẳng.

Hình 6.1 minh họa một số đồ thị : G_1, G_2 là các đồ thị có hướng ; G_3, G_4 là các đồ thị vô hướng. Ở đây, ta quy ước : các đỉnh được đánh số từ 1 đến n (nếu đồ thị có n đỉnh).

Nếu trên đồ thị G (V, E) có một cung đi từ u tới v thì ta nói : v là đỉnh lân cận của u, hay là đỉnh *kề* (adjacent) của u. Tất nhiên với đồ thị vô hướng, khi có cung giữa 2 đỉnh, thì ta nói đỉnh này là lân cận của đỉnh kia.



HÌNH 6.1

Một *đường đi* (path) từ đỉnh u tới đỉnh v là một dãy các đỉnh :

$x_o, x_1, ..., x_{n-1}, x_n$ (n là số nguyên dương), trong đó : $u = x_o$, $v = x_n$ và (x_i, x_{i+1}) với $i = 0, 1, 2, ..., n - 1$ là các cung thuộc E. u gọi là *đỉnh đầu*, v gọi là *đỉnh cuối*. Số lượng các cung trên một đường đi được gọi là *độ dài của đường đi* (path length).

Một *đường đi đơn* (simple path) là đường đi mà mọi đỉnh trên đó, trừ đỉnh đầu và đỉnh cuối, đều khác nhau.

Trường hợp đỉnh đầu trùng với đỉnh cuối ta gọi là *chu trình* (cycle).

Ví dụ : Với đồ thị G_2 (hình 6.1), ta có :

Đường đi 1, 2, 3, 4 là đường đi đơn có độ dài bằng 3.

Đường đi 1, 4, 2, 3, 1 là một chu trình có độ dài bằng 4.

Một đồ thị gọi là *liên thông* (connected) nếu luôn có đường đi giữa 2 đỉnh bất kì của nó. Đối với đồ thị có hướng thì trường hợp đó người ta gọi là *liên thông mạnh* (strongly connected).

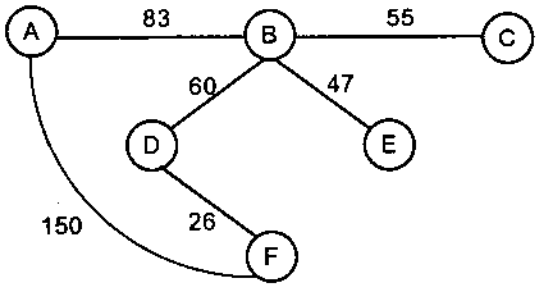
Ví dụ : trong hình 6.1 :

– Các đồ thị G_2 và G_3 là liên thông

– Các đồ thị G_1 và G_4 là không liên thông (vì ở G_1 không có đường đi, chẳng hạn, từ đỉnh 1 đến đỉnh 2 ; ở G_4 không có đường đi từ đỉnh 4 đến đỉnh 3 v.v...).

Có khi ở mỗi cung của đồ thị người ta gắn vào đấy một giá trị thể hiện một thông tin nào đó có liên quan tới cung, giá trị đó được gọi là *trọng số*, và đồ thị được gọi là *đồ thị có trọng số* (weighted graph).

Ví dụ : đồ thị trên hình 6.2 thể hiện các đường đi nối giữa 6 tỉnh A, B, C, D, E, F với các trọng số là độ dài, tính theo km, của các tuyến đường.



HÌNH 6.2

6.2. BIỂU DIỄN TRONG MÁY CỦA ĐỒ THỊ

Cũng như các cấu trúc dữ liệu nói ở các chương trước, đối với đồ thị, ta cũng xét tới 2 cách biểu diễn thông dụng sau đây :

6.2.1. Biểu diễn bằng ma trận lân cận (lưu trữ kế tiếp)

Xét một đồ thị có hướng G (V, E) với V có n đỉnh ($n \geq 1$). Giả sử các đỉnh đã được đánh số từ 1 đến n.

Đồ thị G có thể được biểu diễn trong máy bởi một ma trận vuông A, kích thước $n \times n$, mà các phần tử A_{ij} của nó sẽ nhận giá trị 1 hoặc 0 :

$A_{ij} = 1$ nếu trên G tồn tại một cung đi từ đỉnh i đến đỉnh j.

$A_{ij} = 0$ nếu không có cung giữa đỉnh i và j.

Dĩ nhiên với đồ thị vô hướng, nếu tồn tại một cung (i, j) thì ta có thể hiểu là tồn tại một cung đi từ i tới j, và ngược lại, cũng tồn tại một cung đi từ j tới i, nghĩa là $A_{ij} = 1$ thì $A_{ji} = 1$.

Như vậy với đồ thị vô hướng thì ma trận A biểu diễn nó có các phần tử đối xứng với nhau qua đường chéo chính.

Ma trận A được gọi là *ma trận lân cận* hay *ma trân kề* (adjacency matrix) của đồ thị G.

Ví dụ : với đồ thị G_2 ở hình 6.1 thì ma trận lân cận có dạng :

$$A = \begin{bmatrix} 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \end{bmatrix}$$

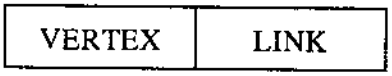
Với đồ thị G_3 ở hình 6.1 thì ma trận lân cận có dạng :

$$A = \begin{bmatrix} 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 \\ 1 & 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 1 & 0 \end{bmatrix}$$

Do ma trận được lưu trữ kế tiếp (đã nêu ở mục 2.2.2 chương 2) nên việc truy cập vào các phần tử của ma trận lân cận được thực hiện trực tiếp, với tốc độ nhanh. Nhưng cách lưu trữ này rõ ràng tốn không gian nhớ, khi n lớn.

6.2.2. Biểu diễn bằng danh sách lân cận (lưu trữ móc nối)

Với cách biểu diễn này mỗi đỉnh sẽ ứng với một danh sách móc nối được gọi là *danh sách lân cận* hay *danh sách kề* (adjacency list). Mỗi nút trong danh sách đó có quy cách :



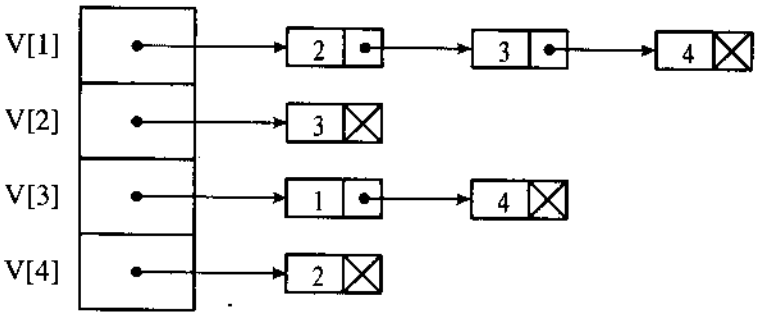
Trường VERTEX chứa số j ứng với đỉnh j là lân cận của đỉnh i ($1 \le i \le n$) trong danh sách lân cận của i.

Trường LINK chứa con trỏ, trỏ tới nút tiếp theo trong danh sách.

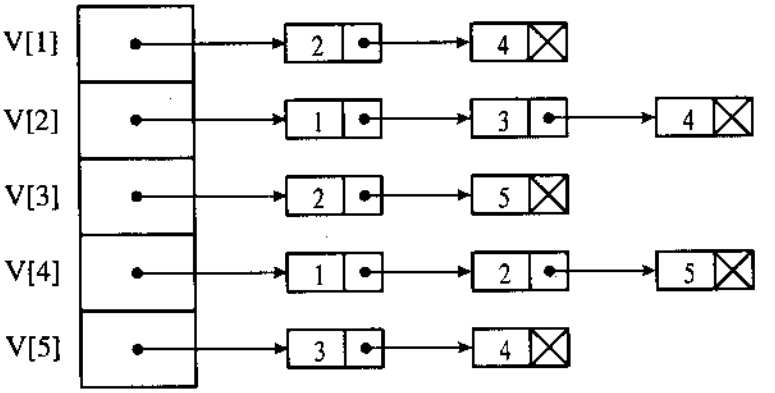
Danh sách sẽ có m nút (người ta còn gọi là "nút danh sách") ; nếu đỉnh i có m đỉnh lân cận.

Mỗi danh sách như vậy lại có "nút đầu danh sách" (list head node). Thường các "nút đầu danh sách" là các phần tử của một vectơ V, có kích thước bằng n, phần tử V[i] của V ($1 \le i \le n$), ứng với danh sách lân cận của nút thứ i, nó chứa địa chỉ nút đầu tiên của danh sách lân cận này.

Hình ảnh danh sách lân cận biểu diễn đồ thị G_2 và G_3 sẽ như sau :



a) Danh sách lân cận biểu diễn G_2



b) Danh sách lân cận biểu diễn G_3

(ở đây ta quy ước : các số ở trường VERTEX của các nút được sắp xếp theo thứ tự tăng dần).

HÌNH 6.3

Ta thấy nếu đồ thị có hướng G (V, E) có n đỉnh, e cung thì số nút cần thiết sẽ là : n "nút đầu danh sách" và e "nút danh sách". Nhưng với đồ thị vô hướng thì lại cần n + 2e nút.

6.3. ÁP DỤNG

6.3.1. Bài toán tìm đường đi

Xét một đồ thị G(V, E) với n đỉnh được đánh số từ 1 tới n.

Một câu hỏi có thể được đặt ra là :

"có hay không đường đi từ đỉnh i tới đỉnh j" (1)

Nếu đồ thị này biểu diễn một mạng lưới đường bộ giữa các thành phố thuộc một vùng nào đó, thì đối với một người đi du lịch bằng xe đạp, muốn đi từ i đến j, câu hỏi này họ sẽ phải đặt ra đầu tiên. Hành trình của họ chỉ có thể tiếp tục được nếu câu trả lời là "có". Mà "có" thì cần phải hiểu là : có thể có nhiều đường, nhưng ít nhất thì cũng phải có một đường.

Vậy để trả lời cho câu hỏi (1) đặt ra ở trên, ta phải dựa vào đâu ?

Ta biết rằng : đồ thị G(V, E) được biểu diễn trong máy bởi ma trận lân cận A. Nếu A_{ij} =1 thì nghĩa là : có một cung đi từ i tới j hay có thể nói : có đường đi độ dài 1 (bằng 1 cung) từ i tới j.

Nhưng ma trận A không đủ để trả lời cho câu hỏi trên, vì đường đi từ i tới j không nhất thiết phải là một cung. Rất có thể không có cung đi từ i tới j, nhưng vẫn có đường đi từ i tới j, qua một số đỉnh khác. Nếu qua một đỉnh

khác thì đường đi đơn sẽ có độ dài 2, nếu qua hai đỉnh khác thì đường đi đơn có độ dài 3, v.v.. Nếu đỉnh i khác đỉnh j thì đường đi đơn dài nhất từ i tới j sẽ có độ dài (n – 1). Còn nếu i trùng với j thì có thể ta có một chu trình với độ dài cực đại là n.

Vì vậy ta cần xét thêm các ma trận mới được xây dựng từ ma trận lân cận A của G(V, E).

Trước hết, ta thấy ; các phần tử của ma trận A chỉ lấy giá trị là 1 hoặc 0 (giá trị lôgích). Vì vậy ta có thể coi ma trận A là một ma trận lôgích.

Bây giờ ta xét tới các phép toán lôgích tác động trên các ma trận lôgích.

Đầu tiên ta nhắc lại hai phép toán lôgích tác động trên các biến lôgích a và b (biến chỉ nhận giá trị 0 hoặc 1). Đó là phép cộng lôgích và phép nhân lôgích.

– Phép cộng lôgích (hay còn gọi là *phép tuyển, phép hoặc*) được kí hiệu là $\vee$ (trong ngôn ngữ lập trình được gọi là *OR*).

– Phép nhân lôgích (hay còn gọi là *phép hội, phép và*) được kí hiệu là $\wedge$ (trong ngôn ngữ lập trình được gọi là *AND*). Quy tắc của chúng được nêu trong bảng sau :

a	b	$a \vee b$	$a \wedge b$
0	0	0	0
0	1	1	0
1	0	1	0
1	1	1	1

Xét hai ma trận lôgích A và B, kích thước $n \times n$.

• Phép cộng lôgích A và B để cho ma trận tổng C, kích thước $n \times n$:

$C = A \vee B$, sẽ được thực hiện bằng cách tính :

$$C_{ij} = A_{ij} \vee B_{ij} \qquad \text{với} \quad 1 \le i \le n, \quad 1 \le j \le n.$$

• Phép nhân lôgích A và B để cho ma trận tích D, kích thước $n \times n$:

$D = A \wedge B$, sẽ được thực hiện bằng cách tính :

$$D_{ij} = \bigvee_{k=1}^{n} (A_{ik} \wedge B_{kj}) \qquad \text{với} \quad 1 \le i \le n, \quad 1 \le j \le n.$$

Với ma trận lân cận A của G đã biết, ta có :

$$A^{(2)} = A \wedge A$$

Ta thấy phần tử $A_{ij}^{(2)}$ của $A^{(2)}$ bằng 1 khi và chỉ khi, ít nhất có một giá trị k để $A_{ik} \wedge A_{kj} = 1$ (vì $A_{ij}^{(2)} = \bigvee_{k=1}^{n} (A_{ik} \wedge A_{kj})$, mà $A_{ik} \wedge A_{kj} = 1$ khi và chỉ khi
có $A_{ik} = 1$ và $A_{kj} = 1$; nghĩa là có một cung đi từ i tới k và có một cung đi từ k tới j. Điều đó chứng tỏ : có một đường đi độ dài 2 từ i tới j.

Như vậy ma trận $A^{(2)}$ sẽ cho ta câu trả lời của câu hỏi : "có hay không đường đi độ dài 2 từ i tới j" :

Nếu $A_{ij}^{(2)} = 1$ thì nghĩa là "có".

Nếu $A_{ij}^{(2)} = 0$ thì nghĩa là "không".

Tương tự như vậy ta có thể xét tới các ma trận :

$$A^{(3)} = A \wedge A^{(2)} ;$$

$$A^{(4)} = A \wedge A^{(3)}$$

$$\vdots$$

$$A^{(r)} = A \wedge A^{(r-1)}$$

và ta cũng suy ra được :

Nếu $A_{ij}^{(r)} = 1$ thì "có" đường đi độ dài r từ i tới j (ít nhất cũng có 1 đường).

Nếu $A_{ij}^{(r)} = 0$ thì "không có".

Từ đó ta thấy rằng : để trả lời cho câu hỏi (1) đặt ra ở trên ta phải lập ma trận :

$$P = A \vee A^{(2)} \vee A^{(3)} \vee \ldots \vee A^{(n)} \qquad (2)$$

$$= \bigvee_{k=1}^{n} A^{(k)}$$

và nếu : $P_{ij} = 1$ thì câu trả lời là "có".

$P_{ij} = 0$ - - "không".

Ma trận P được gọi là *ma trận đường đi* (path matrix) của G.

Việc tính P theo công thức (2) sẽ tốn nhiều thời gian khi n lớn. Giải thuật của warshall nêu sau đây, có hiệu lực cao hơn.

Procedure WARSHALL (A, n, P) ;

1. **for** i := 1 **to** n **do**

    for j : = 1 **to** n **do**

        P[i, j] := A[i, j] ;

2. **for** k := 1 **to** n **do**

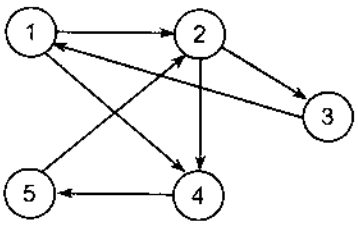
    for i := 1 **to** n **do**

        for j := 1 **to** n **do**

            P[i, j] := P[i, j] ∨ (P[i, k] ∧ P[k, j])

3. **return**

Ví dụ : Cho đồ thị G với ma trận lân cận A có dạng như sau :



$$A = \begin{bmatrix} 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 \end{bmatrix}$$

HÌNH 6.4

Ta xét việc tính $A^{(2)}$. Biết rằng mỗi phân tử $A_{ij}^{(2)}$ được tính theo công thức : $A_{ij}^{(2)} = \bigvee_{k=1}^{5} (A_{ik} \wedge A_{kj})$ với $1 \le i, j \le 5$. Chẳng hạn :

$$A_{11}^{(2)} = A_{11} \wedge A_{11} \vee A_{12} \wedge A_{21} \vee A_{13} \wedge A_{31} \vee A_{14} \wedge A_{41} \vee A_{15} \wedge A_{51}$$

$$= 0 \wedge 0 \vee 1 \wedge 0 \vee 0 \wedge 1 \vee 1 \wedge 0 \vee 0 \wedge 0 = 0$$

$$A_{12}^{(2)} = 0 \wedge 1 \vee 1 \wedge 0 \vee 0 \wedge 0 \vee 1 \wedge 0 \vee 0 \wedge 1 = 0...$$

cuối cùng, ta có ;

$$A^{(2)} = \begin{bmatrix} 0 & 0 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 \end{bmatrix}$$

Tương tự ta cũng tính được :

$$A^{(3)} = \begin{bmatrix} 1 & 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 0 & 1 \end{bmatrix} \qquad A^{(4)} = \begin{bmatrix} 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 \end{bmatrix}$$

$$A^{(5)} = \begin{bmatrix} 1 & 0 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 & 1 \end{bmatrix} \qquad P = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \end{bmatrix}$$

Từ kết quả trên có thể nêu lên một số nhận xét, ví dụ :

$A^{(2)}[1,5] = 1$ nghĩa là : có đường đi độ dài 2 từ 1 tới 5 (không có cung từ 1 tới 5). Kiểm tra trên đồ thị ta thấy đó chính là đường đi 1, 4, 5.

$A^{(3)}[4,3] = 1$ nghĩa là : có đường đi độ dài 3 từ 4 tới 3. Đó chính là đường đi 4, 5, 2, 3.

$A^{(4)}[3,2] = 1$ nghĩa là : có đường đi độ dài 4 từ 3 đến 2. Đó chính là đường đi 3, 1, 4, 5, 2.

$A^{(5)}[5,5] = 1$ nghĩa là : có chu trình độ dài 5, tại đỉnh 5. Đó chính là chu trình 5, 2, 3, 1, 4, 5.

Các phần tử của P đều bằng 1 chứng tỏ từ bất kì một đỉnh nào trên đồ thị G cũng có đường đi tới một đỉnh khác. Nói một cách khác G là đồ thị liên thông mạnh. Hơn nữa : bất kì đỉnh nào trên đồ thị G cũng là đỉnh đầu và đỉnh cuối của ít nhất một chu trình.

6.3.2. Bài toán sắp xếp Tô – pô

Ta hãy xét một bài toán cụ thể :

Có n công việc ; giả sử, để phân biệt, ta đánh số từ 1 đến n. Giữa các công việc này có một quan hệ mà ta gọi là quan hệ "làm trước" và kí hiệu bởi dấu <. Nếu i < j thì điều đó nghĩa là : công việc i phải được thực hiện trước công việc j. Chẳng hạn công việc "làm sắt và đóng cốp pha" phải được thực hiện trước công việc "đổ bê tông".

Một đội thợ phải thực hiện các công việc này kế tiếp nhau (nghĩa là : làm xong việc này rồi mới làm đến việc khác) trong một khoảng thời gian nào đó. Vậy thì người đội trưởng sẽ phải sắp xếp các công việc đó theo một thứ tự tuyến tính sao cho khi thực hiện lần lượt các công việc theo thứ tự đó, quan hệ "làm trước" giữa các công việc không bị phá vỡ. Điều đó có nghĩa là : các công việc sẽ được thực hiện tuần tự và việc cần làm trước phải được thực hiện trước, việc phải làm sau sẽ được thực hiện sau, để không bao giờ xảy ra tình trạng vô lí như kiểu : điều thợ đi đổ bê tông khi chưa cho làm sắt và đóng cốp pha.

Giả sử có 8 công việc được đánh số từ 1 đến 8 mà :

$2 < 3$ $\quad 4 < 3$ $\quad 6 < 1$

$2 < 4$ $\quad 4 < 5$ $\quad 7 < 2$

$3 < 6$ $\quad 5 < 6$ $\quad 8 < 2$

Nếu người đội trưởng sắp xếp các công việc đó theo trình tự :

8, 7, 2, 4, 5, 3, 6, 1 (3)

hoặc

7, 8, 2, 4, 3, 5, 6, 1 (4)

thì việc thực hiện sẽ hoàn toàn thuận lợi vì nếu theo (3) :

– Khi công việc 2 được thực hiện thì các công việc cần phải làm trước như công việc 7, công việc 8 đã được thực hiện rồi.

– Khi công việc 4 thực hiện thì công việc 2 đã thực hiện rồi.

– Khi công việc 3 thực hiện thì công việc 2 và 4 đã thực hiện rồi.

– Khi công việc 5 thực hiện thì công việc 4 đã thực hiện rồi.

– Khi công việc 6 thực hiện thì công việc 3 và 5 đã thực hiện rồi.

Một thứ tự như (3) hoặc (4), được gọi là *thứ tự tôpô* (topological order), việc sắp xếp như trên gọi là *sắp xếp tô pô* (Topological sort).

Ta cũng thấy : các công việc với quan hệ "làm trước", kí hiệu bằng dấu $<$, luôn thỏa mãn các tính chất sau :

a) Nếu $x < y$, $y < z$ thì $x < z$ (tính bắc cầu).

b) Nếu $x < y$ thì không tồn tại $y < x$ (tính không đối xứng).

c) Không bao giờ tồn tại $x < x$ (tính không phản xạ).

Người ta gọi một quan hệ thỏa mãn 3 tính chất trên là một "*thứ tự bộ phận*" (partial order). Một tập hợp S mà giữa các phần tử có một quan hệ nào đó (thường được gọi theo một tên chung là quan hệ "trước") thỏa mãn 3 tính chất đã nêu ở trên thì S được gọi là : *tập có thứ tự bộ phận*.

Bài toán sắp xếp tô pô là bài toán được đặt ra với một tập S có thứ tự bộ phận. Nó chính là bài toán : "sắp xếp các phần tử của S theo một thứ tự tuyến tính sao cho thứ tự bộ phận vẫn được đảm bảo".

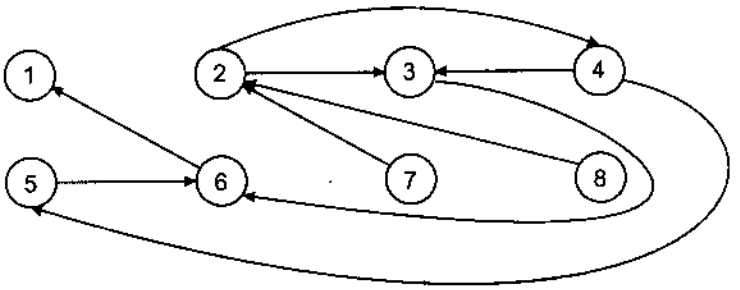
Ta có thể biểu diễn tập S, có thứ tự bộ phân bằng một đồ thị định hướng, theo quy tắc sau :

– Mỗi phân tử của S là một đỉnh.

– Phần tử i "trước" phân tử j sẽ ứng với một cung có hướng (i, j) trên đồ thị.

Rõ ràng, do tính chất của tập có thứ tự bộ phận, trên đồ thị định hướng này không bao giờ tồn tại một chu trình (vì nếu có chu trình thì tính chất 3, trong 3 tính chất đã nêu ở trên sẽ bị vi phạm).

Như vậy tập của 8 công việc nêu ở trên, chính là một tập có thứ tự bộ phận và nó được biểu diễn bởi một đồ thị định hướng như hình 6.5 :



HÌNH 6.5

Như vậy sắp xếp tô pô đối với tập các công việc này chính là sắp xếp các đỉnh của đồ thị nói trên thành một dãy theo hàng ngang sao cho các cung (mũi tên) đều hướng từ trái sang phải.

Có thể thực hiện được điều này bằng *quy tắc chung* như sau :

– Đầu tiên chọn ra một đỉnh không có cung nào đi tới nó (không có mũi tên đâm vào nó) và đưa đỉnh đó ra sắp xếp.

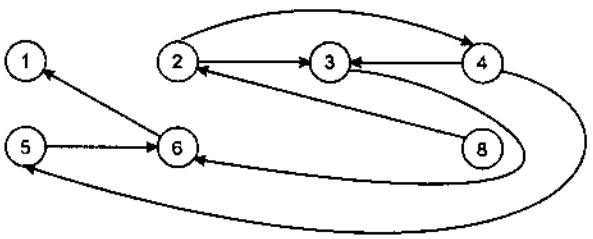
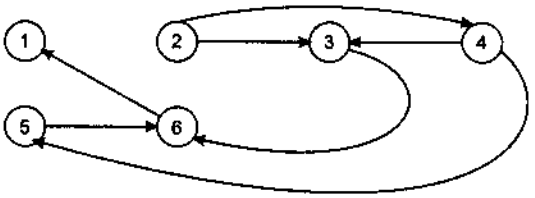
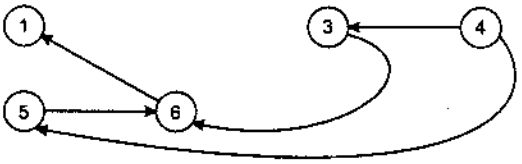
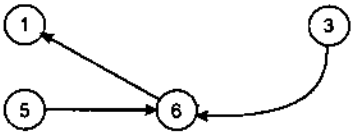
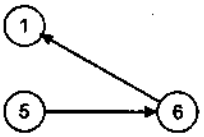
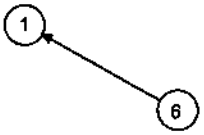
– Loại đỉnh này ra khỏi đồ thị cùng với các cung xuất phát từ nó. Vì nó đã trở thành "trước" của các đỉnh lân cận của nó rồi nên cung thể hiện quan hệ này không cần nữa.

– Đồ thị với các đỉnh còn lại, lại được thực hiện tương tự như trên và cứ như thế cho tới khi các đỉnh đã được đưa ra sắp xếp theo thứ tự tuyến tính.

Trường hợp cùng một lúc nếu có nhiều đỉnh đều không có cung nào đi tới nó thì có thể chọn một đỉnh tùy ý.

Như vậy nghĩa là : có thể có nhiều thứ tự tô pô khác nhau.

Có thể diễn tả quá trình thực hiện "quy tắc sắp xếp tô pô" nêu trên, với đồ thị hình 6.5 như hình 6.6 :

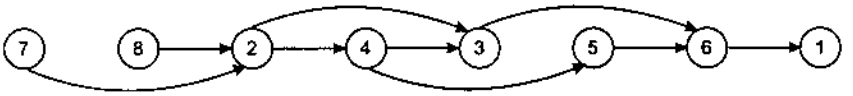
Đỉnh đưa ra sắp xếp	Phần đồ thị còn lại
7	
8	
2	
4	
3	
5	
6	1
1	

HÌNH 6.6

Thoạt đầu ta thấy đỉnh 7 và 8 đều không có cung nào đi tới nó. Giả sử ta chọn 7 để đưa ra sắp xếp trước tiên.

Như vậy ta đã có một thứ tự tô pô giống như dãy (4) đã nêu ở trên.

Bây giờ đồ thị với các đỉnh được sắp xếp theo thứ tự tuyến tính, sẽ có dạng như hình 6.7.



HÌNH 6.7

Rõ ràng : ở đây các cung đi từ trái sang phải, nghĩa là theo thứ tự tuyến tính này thì thứ tự bộ phận đã được đảm bảo, hay nói một cách khác khi thực hiện tuần tự các công việc như trên việc nào cần làm trước đều đã được làm trước.

Việc xây dựng một giải thuật sắp xếp tô pô sẽ đòi hỏi ta phải biết được :

– Số các cung đi tới một đỉnh. Nếu số này bằng 0 thì đỉnh đó có thể đưa ra sắp xếp được.

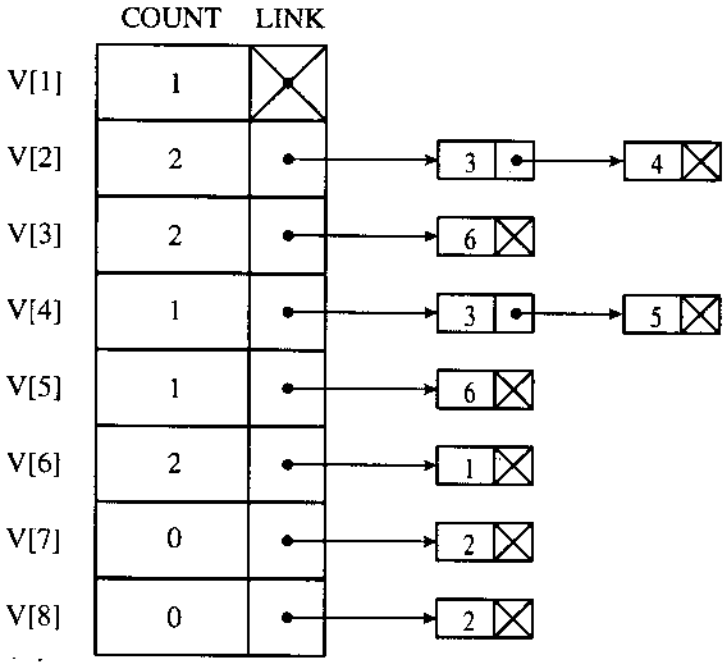
– Các cung đi ra từ một đỉnh i, nghĩa là biết các đỉnh lân cận của i. Để khi một đỉnh i nào đó đã được đưa ra sắp xếp thì số lượng cung đi vào đỉnh lân cận j của nó sẽ được giảm đi 1 (ứng với việc loại cung xuất phát từ đỉnh i).

Các yêu cầu này sẽ được thỏa mãn nếu ta cài đặt đồ thị bằng danh sách lân cận, với quy ước bổ sung như sau :

Nút "đầu danh sách" của danh sách lân cận ứng với đỉnh i, ngoài trường LINK, như thông thường còn có thêm trường COUNT mà COUNT (i) ghi nhận số các cung đi tới đỉnh i. Như với đồ thị hình 6.5 thì cấu trúc lưu trữ của nó có dạng như hình 6.8.

Dĩ nhiên phải có một chương trình tạo lập nên cấu trúc này ở trong máy, trước khi thực hiện giải thuật sắp xếp tô pô.

Giải thuật TOPO – ORDER sau đây sẽ thực hiện sắp xếp tô pô cho tập S có thứ tự bộ phận với S được biểu diễn bởi đồ thị mà cấu trúc lưu trữ của nó có dạng danh sách lân cận như vừa nêu ở trên. Trong giải thuật này người ta dùng một queue Q để lưu trữ các đỉnh i mà COUNT (i) = 0.

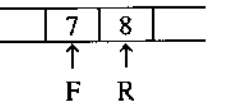
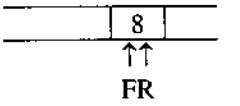
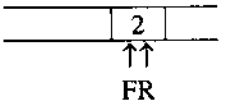
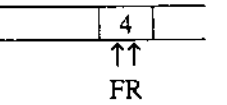
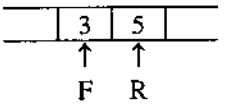
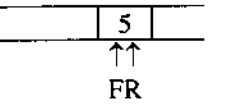
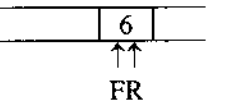
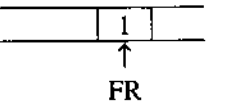
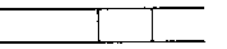


HÌNH 6.8

Procedure TOPO – ORDER (COUNT, VERTEX, LINK, n) ;

1. **for** i := 1 **to** n **do**
   if count (i) = 0 **then** nạp i vào Q ;
2. **Repeat**
3. Đưa đỉnh j ở "lối trước" của Q ra sắp xếp ;
4. ptr := LINK(j) ; {tìm đến đỉnh lân cận đầu tiên của j}
5. **While** ptr ≠ null **do begin**
6. k := VERTEX (ptr) ; {k là đỉnh lân cận đầu tiên của j}
7. COUNT (k) := COUNT (k) – 1 ;
8. **if** COUNT (k) = 0 **then** nạp k vào Q
9. ptr := LINK (ptr) {tìm đến đỉnh lân cận khác của j}
   end
   until Q rỗng ;
10. **return**

Nếu áp dụng giải thuật TOPO – ORDER để sắp xếp tô pô cho tập các công việc mà đồ thị biểu diễn nó được cài đặt trong máy bởi danh sách lân cận ở hình 6.8 thì diễn biến qua từng bước có thể minh họa như sau :

Đỉnh i được đưa ra sắp xếp	COUNT (k) lân cận của i	Tình trạng của queue Q
Thoạt đầu các đỉnh i mà COUNT (i) = 0 được nạp vào Q		 F : trỏ tới lối trước R : trỏ tới lối sau
(7)	COUNT(2) = 1	
(8)	COUNT (2) = 0	
(2)	COUNT (3) = 1 COUNT (4) = 0	
(4)	COUNT (3) = 0 COUNT (5) = 0	
(3)	COUNT (6) = 1	
(5)	COUNT (6) = 0	
(6)	COUNT (1) = 0	
(1)		 queue rỗng

Cuối cùng ta có thứ tự tô pô – minh họa như ở hình 6.7.

Chú ý : 1. Với giải thuật TOPO – ORDER ta chỉ nhận được duy nhất một thứ tự tôpô thôi.

2. Nếu ta lưu trữ các đỉnh i mà COUNT (i) = 0 bằng một stack thì thứ tự tô pô nhận được tất nhiên sẽ khác.

3. Có nhiều bài toán khác dẫn tới bài toán sắp xếp tôpô.

Câu hỏi và bài tập

6.1. Hãy nêu vài ví dụ thực tế biểu diễn được bằng cấu trúc đồ thị.

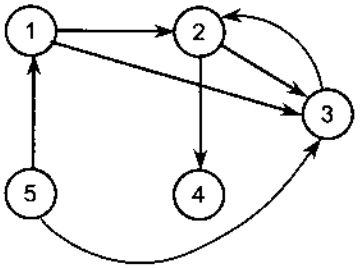
6.2. Cây có phải là một đồ thị không ?

6.3. Một đồ thị vô hướng có 5 đỉnh thì có tối đa bao nhiêu cung giữa một đỉnh với một đỉnh khác.

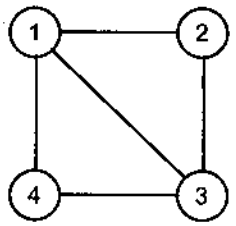
Với đồ thị có n đỉnh thì sao ?

6.4. Vẽ một đồ thị định hướng có 5 đỉnh A, B, C, D, E và các cung định hướng : (A, B) ; (C, D) ; (D, A) ; (B, D) ; (B, E) ; (E, D).

6.5. Cho các đồ thị G_1 và G_2 sau đây



G_1

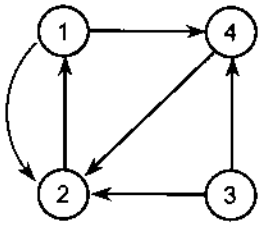


G_2

a) Hãy lập ma trận lân cận của G_1 và G_2

b) Hãy lập danh sách lân cận của G_1 và G_2

6.6. Cho đồ thị G_3. Hãy lập ma trận lân cận A của G_3. Tính $A^{(2)}$, $A^{(3)}$, $A^{(4)}$ và ma trận đường đi P. Nêu ý nghĩa của P, cho ví dụ minh họa.



G_3

6.7. Cho đồ thị G_2 như ở bài tập 6.5. Gọi A là ma trận lân cận của G_2.

a) Hãy tính A[2, 4], $A^{(2)}$ [2, 4] và giải thích sự khác nhau giữa chúng.

b) Hãy tính $A^{(3)}$[1, 4] và nêu ý nghĩa của nó.

6.8. Một đội thợ phải thực hiện lần lượt 6 công việc khác nhau. Các công việc này được đánh số từ 1 đến 6 và giữa chúng có quan hệ "làm trước", kí hiệu bởi dấu (<). Biết rằng :

1 < 2	4 < 2	6 < 5
2 < 3	4 < 6	
2 < 5	6 < 3	

a) Hãy biểu diễn tập các công việc này, với quan hệ < đã nêu trên, bằng một đồ thị định hướng. Vẽ đồ thị đó.

b) Minh họa dạng của đồ thị với các đỉnh được xét trong quá trình sắp xếp tô pô theo quy tắc chung. Nêu cụ thể 2 thứ tự tô pô khác nhau.

c) Nêu áp dụng giải thuật TOPO – ORDER thì đồ thị nêu ở câu a) được cài đặt trong máy có dạng như thế nào, trước khi thực hiện giải thuật.

d) Sau khi thực hiện giải thuật TOPO – ORDER thì thứ tự Tôpô nhận được có dạng thế nào ?

6.9. Một tập S gồm 7 công việc (được đánh số từ 1 đến 7) với quan hệ "làm trước", thỏa mãn các điều kiện sau :

7 < 5	3 < 6	2 < 1	4 < 2
5 < 3	5 < 6	2 < 6	

a) Hãy vẽ đồ thị biểu diễn S.

b) Hãy minh họa cấu trúc lưu trữ của đồ thị đó trước khi thực hiện giải thuật TOPO – ORDER.

c) Nêu kết quả sau khi thực hiện giải thuật TOPO – ORDER.

HƯỚNG DẪN GIẢI BÀI TẬP

Ở đây sẽ nêu lên phần hướng dẫn giải hoặc lời giải của một số bài tập trong từng chương.

CHƯƠNG 1

1.4. Giải thuật mà độ phức tạp về thời gian của nó là O(1), nếu thời gian thực hiện nó chỉ bằng một hằng số.

Ví dụ : giải thuật tính và in X^2

```
Program BINH – PHUONG ;
    Read (X) ;
    Y := X * X ;
    write Y ;
    return
```

1.7. a) $T(n) = n^3 + 100\ n\log_2 n + 5000$

Ta thấy : nếu n = 32 thì $n^3 = 32768$

$$\log_2 n = 5$$

$$100\ n\log_2 n = 16000 < n^3$$

Do đó : $T(n) < n^3 + n^3 + n^3 = 3n^3$, nếu n > 32. Rõ ràng tồn tại một hằng số C = 3 và một giá trị $n_o = 32$ để khi $n \geq n_o$ ta luôn có :

$$T(n) < Cn^3$$

Vậy ta có thể viết :

$$T(n) = O(n^3)$$

Tương tự ta cũng xác nhận được :

b) $T(n) = O(2^n)$; c) $T(n) = O(n^2)$; d) $T(n) = O(\log_2 n)$.

1.8. a) Ta chọn phép toán :

$$A[i, j] := B[i, j] + C[i, j]$$

làm phép tích cực.

Với mỗi giá trị của i, phép này đều thực hiện n lần, mà i lấy giá trị từ 1 đến n nên phép toán trên thực hiện tổng cộng n × n lần.

Do đó có thể viết : $T_1(n) = Cn^2$

hay $T_1(n) = O(n^2)$.

b) $T_2(n) = O(n)$; c) $T_3(n) = O(n^3)$; d) $T_4(n) = O(n)$.

1.10. a) Tiêu chuẩn gốc ở đây là m = 0, nếu m = 0 thì Acker (m,n) = n + 1

b) Ta có :

Acker (1, 3) = Acker (0, Acker (1, 2))

mà Acker (1, 2) = Acker (0, Acker (1, 1))

Acker (1, 1) = Acker (0, Acker (1, 0))

Acker (1, 0) = Acker (0, 1) = 2

Từ đó suy ra :

Acker (1, 3) = 5

1.11. a) $F_{16} = 987$

b) **Procedure** FIBONACCI (n, F) ;

{ở đây F là một vectơ có n phần tử để lưu trữ n số Fibonacci đầu tiên}

1. {Tạo lập 2 số Fibonacci đầu tiên}

   F[1] := 1 ; F[2] := 1 ;

2. {Tính lần lượt các số Fibonacci tiếp theo}

   for i := 3 **to** n **do**

   F[i] := F[i – 1] + F[i – 2] ;

3. {In lần lượt n số Fibonacci}

   for i := 1 **to** n **do**

   write (F[i]) ;

4. **return**.

1.12. a) Các giá trị của p, q, r được ghi nhận trong bảng sau :

p	q	r
1260	198	72
198	72	54
72	54	18
54	18	0

cuối cùng ta có : USCLN (1260, 198) = 18.

b) Ta thấy việc tính USCLN của p và q sẽ dẫn tới việc tính USCLN của q và r khi r ≠ 0 (q và r rõ ràng là nhỏ hơn p và q).

c) Giải thuật đệ quy :

Function RUSCLN (p, q) ;

{Chú ý là số dư r của p và q được xác định bởi phép tính p mod q}

1. **if** p mod q = 0 **then** RUSCLN ; = q
2. **else** RUSCLN := RUSCLN (q, p mod q) ;
3. **return**

• Giải thuật lặp

Function IUSCLN (p, q) ;

{x, y, z ở đây là các biến cục bộ}

1. x := p ; y := q ;
2. **repeat** z := x mod y ;
   x := y ;
   y := z
   until z : = 0 ;
3. IUSCLN := x ;
4. **return.**

CHƯƠNG 2

2.2. *Chú ý :* Chương trình dịch của ngôn ngữ PASCAL thực hiện lưu trữ mảng theo thứ tự ưu tiên hàng. A ở đây là một ma trận có 5 hàng, 4 cột, nó sẽ được cài đặt bằng một vectơ có 5 × 4 = 20 phần tử.

2.3. Đối với mảng n chiều (n ≥ 3) thì :

– "Thứ tự ưu tiên hàng" tương ứng với việc các phần tử của mảng được liệt kê sao cho chỉ số cuối cùng được biến đổi trước, rồi lần lượt tới các chỉ số tiếp theo, theo chiều từ phải qua trái.

– "Thứ tự ưu tiên cột" tương ứng với việc các phần tử của mảng được liệt kê sao cho chỉ số đầu tiên được biến đổi trước, rồi lần lượt tới các chỉ số tiếp theo, theo chiều từ trái qua phải.

Với mảng 3 chiều B mà phần tử của nó là B[i, j, k] với 1 ≤ i ≤ 2 ; 1 ≤ j ≤ 4 ; 1 ≤ k ≤ 3 thì sẽ được cài đặt bởi một vectơ có 2 × 4 × 3 = 24 phần tử.

a) Theo thứ tự ưu tiên hàng :

B[1, 1, 1] ; B[1, 1, 2] ; B[1, 1, 3] ; B[1, 2, 1] ; B[1, 2, 2] ; B[1, 2, 3] ;
B[1, 3, 1] ; B[1, 3, 2] ; B[1, 3, 3] ; B[1, 4, 1] ; B[1, 4, 2] ; B[1, 4, 3] ;
B[2, 1, 1] ; B[2, 1, 2] ; B[2, 1, 3] ; B[2, 2, 1] ; B[2, 2, 2] ; B[2, 2, 3] ;
B[2, 3, 1] ; B[2, 3, 2] ; B[2, 3, 3] ; B[2, 4, 1] ; B[2, 4, 2] ; B[2, 4, 3].

b) Theo thứ tự ưu tiên cột :

B[1, 1, 1] ; B[2, 1, 1] ; B[1, 2, 1] ; B[2, 2, 1] ; B[1, 3, 1] ; B[2, 3, 1] ;
B[1, 4, 1] ; B[2, 4, 1] ; B[1, 1, 2] ; B[2, 1, 2] ; B[1, 2, 2] ; B[2, 2, 2] ;
B[1, 3, 2] ; B[2, 3, 2] ; B[1, 4, 2] ; B[2, 4, 2] ; B[1, 1, 3] ; B[2, 1, 3] ;
B[1, 2, 3] ; B[2, 2, 3] ; B[1, 3, 3] ; B[2, 3, 3] ; B[1, 4, 3] ; B[2, 4, 3].

2.7. a) Số phần tử của mảng một chiều (vectơ) được tính theo công thức sau :

$$n = UB - LB + 1$$

với : UB là chỉ số trên (biên trên),

LB là chỉ số dưới (biên dưới).

vậy với vectơ AA thì $n_A = 46$,

với vectơ BB thì $n_B = 16$.

b) LOC (AA[17]) = 300 + 4(17 – 5) = 348

LOC (AA[35]) = 420.

2.8. LOC (CC [4, 2]) = 1000 + 3[(4 – 1)*3 + (2 – 1)]

= 1030

2.9. Vectơ B có kích thước là :

$$1 + 2 + 3 + ... + n = \frac{1}{2}n(n + 1)$$

Từ đó suy ra số ô nhớ tiết kiệm được.

Ta thấy ở đây các phần tử A[i, j] với $1 \le i \le n$, $1 \le j \le n$ sẽ có giá trị bằng 0 nếu $j > i$.

Còn nếu A[i, j] mà $j \le i$ thì nó được lưu trữ bởi phần tử B[K] của vectơ B.

Cần chú ý là : K sẽ bằng tổng số các phần tử khác không của A thuộc (i – 1) hàng đứng trước hàng i, và các phần tử khác không của A đứng trong hàng i tới cột j. Vì vậy :

$$K = \frac{i(i-1)}{2} + j$$

Có thể viết :

$$A[i, j] = \begin{cases} 0 & \text{nếu } j > i \\ B\left[\frac{i(i-1)}{2} + j\right] & \text{nếu } j \le i \end{cases}$$

2.10. Phần xử lí trong giải thuật có thể nêu như sau :

a)

1. **for** i := 1920 **to** 1970 **do**
   if N[i] = 0 **then write** (i) ;
2. **return**

c) 1. N50 : = 0 ;

2. **for** i := 1920 **to** 1934 **do**
   N50 := N50 + N[i]
3. **return**

2.12. a) Sắp xếp theo thứ tự tăng dần thì số nhỏ bao giờ cũng đứng trước số lớn.

Vì vậy nếu duyệt dãy số theo kiểu "top down" thì sẽ thực hiện đổi chỗ khi gặp 2 số kề nhau mà số lớn lại đứng trước số nhỏ.

Sau đây là bảng minh họa đối với dãy số cho ở bài tập 2.11.

Lượt	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]	A[8]
	73	33	44	11	88	22	66	55
1	33	44	11	73	22	66	55	88
2	33	11	44	22	66	55	73	88
3	11	33	22	44	55	66	73	88
4	11	22	33	44	55	66	73	88
5	–	–	–	–	–	–	–	–
6	–	–	–	–	–	–	–	–

ta thấy từ lần thứ 5 trở đi không còn phép đổi chỗ nào nữa. Theo cách làm này các số lớn cũng "nổi dần lên" nếu xét theo chiều từ A[1] đến A[8] và sau lượt thứ i thì số phần tử được xét chỉ còn (n – i) thôi.

b) **Procedure** BBS (A, n) ;

1. **for** i := 1 **to** n – 1 **do**
2. j := 1 ;
3. **while** j ≤ n – i **do begin**
   if A[j] > A[j + 1] **then**
   A[j] ↔ A[j + 1] ;
   j := j + 1
   end
4. **return**

2.13. Phải thực hiện 3 lượt

3.5.

a) Chỉ cần thay đổi phép xử lí ở bước 3 của giải thuật TRAVERS (L) (trong bài) là ta có giải thuật thực hiện được yêu cầu đề ra.

b) **Function** COUNT (LIST) ;

1. {Khởi tạo biến đếm n và biến trỏ p} n:= 0; p:= LIST;

2. {duyệt qua danh sách (nếu danh sách không rỗng) và tăng giá trị của biến đếm lên 1 mỗi khi p trỏ tới một nút mới trong danh sách).

```
while p ≠ null   do begin
                        n := n + 1 ;
        p := LINK (p) ;
                                {p trỏ tới nút tiếp theo}
                        end ;
```

3. COUNT := n ;

return

{có thể viết gọn bằng câu lệnh **return**(n)}

* Ta cũng có thể thực hiện bằng một giải thuật đệ quy :

Function RCOUNT (LIST)

1. **if** LIST = null **then** RCOUNT := 0
2. **else**
   RCOUNT := RCOUNT (LINK (LIST)) + 1
3. **return**

c) Tương tự như giải thuật ở b) chỉ khác ở chỗ : khi kiểm tra thấy INFO (p) > 0) thì mới thực hiện phép đếm.

3.6. b) Chú ý là có hai tình huống sẽ xảy ra :

1. T đang trỏ tới nút đầu tiên của danh sách. (Danh sách không rỗng nên ít nhất nó cũng có 1 nút và trong trường hợp này thì T cũng trỏ tới nút đó).

2. T trỏ tới một nút không phải là nút đầu tiên, nghĩa là có nút đứng trước nút T. Sau đây là giải thuật :

Procedure INS (P, T, X) ;

1. {Tạo lập nút mới để bổ xung}

call New (R) ;
INFO (R) := X ;

2. {Trường hợp T là nút đầu tiên}

if T = P **then begin**

LINK (R) := P ;

P := R ; {Bổ sung nút R vào và nó trở thành nút đầu tiên}

return

end ;

3. {Trường hợp T không phải là nút đầu tiên}

Q := P ; {Q là một biến trỏ để ghi nhận địa chỉ các nút thuộc danh sách P}

while LINK(Q) ≠ T **do** Q := LINK (Q) ;

{sau khi thực hiện vòng lặp này thì Q sẽ trỏ tới nút trước nút trỏ bởi T}

4. {Chèn nút R vào giữa nút Q và nút T}

LINK (R) := T ;

LINK (Q) := R ;

return

3.7. a) Chú ý tới 2 trường hợp :

1. Danh sách chỉ có một nút thì sau phép loại bỏ nó sẽ trở thành danh sách rỗng (Q := null).

2. Danh sách có nhiều hơn một nút thì sau phép loại bỏ, con trỏ Q sẽ trỏ tới nút thứ hai (Q := LINK(Q)).

Tuy nhiên hai trường hợp này có thể xử lí chung được.

b) Trước hết phải tìm đến nút đứng trước nút T (xem giải thuật INS trong lời giải bài 3.6b) và thay đổi mối nối ở trường LINK của nút này.

3.8. Có thể áp dụng cách tìm đến nút cuối cùng của một danh sách và cách ghép nối như trong giải thuật IN–LIST thuộc mục 3.3.2 trong bài để xây dựng hai giải thuật ứng với câu a) và câu b).

3.9. Nếu Q = P thì không thể tách thành hai danh sách được ;

Nếu Q ≠ P thì phải tìm đến nút trước nút Q. Nút này chính là nút cuối của danh sách thứ nhất.

Procedure TACH (P, Q) ;

1. **if** Q = P **then begin**

write (' không tách thành 2 danh sách được') ;

return

end ;

2. R := P ;

   while LINK (R) ≠ Q **do** R := LINK(R) ;

3. LINK (R) := null ;

4. **return**

3.10. Ở đây ngoài một biến để ghi nhận số lượng các nút trong danh sách (như giải thuật COUNT(LIST) trong bài tập 3.5), ta còn phải dùng thêm một biến nữa để ghi nhận tổng số chứa trong trường INFO của các nút thuộc danh sách.

```
        Procedure TB (LIST, M) ;
1.      SUM := 0 ; n ;= 0 ; p := LIST ;
2.      while p ≠ null do begin
                    SUM := SUM + INFO (p) ;
                        n := n + 1 ;
                        p ;= LINK (p)
                        end ;
3.      M := SUM/n ;
4.      return
```

CHƯƠNG 4

4.3. Việc tính giá trị của biểu thức tiền tố cũng được thực hiện tương tự như đối với biểu thức hậu tố, chỉ có khác ở chỗ :

– Khi đọc biểu thức ta phải thực hiện từ phải qua trái.

– Khi đọc một toán tử (phép toán) thì 2 toán hạng vừa đọc sát nó sẽ được kết hợp với toán tử đó để tạo nên toán hạng mới ứng với toán tử sẽ đọc sau đó.

Ví dụ : Quá trình tính giá trị của biểu thức tiền tố :

* A – B + C D

ứng với A = 2 ; B = 9 ; C = 4 ; D = 3 sẽ được thực hiện như sau :

Khi đọc phép "+" thì thực hiện C + D, nghĩa là 4 + 3 = 7.

Khi đọc phép trừ "–" thì thực hiện B – (C + D), nghĩa là 9 – 7 = 2.

Khi đọc phép nhân "*" thì thực hiện A*(B – (C + D)), nghĩa là 2 * 2 = 4

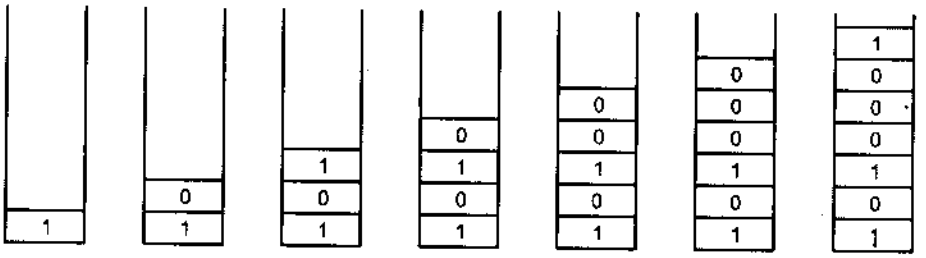
Cuối cùng ta có kết quả bằng 4.

4.4. Dựa vào những điều đã nêu trong mục 4.6 của chương 4.

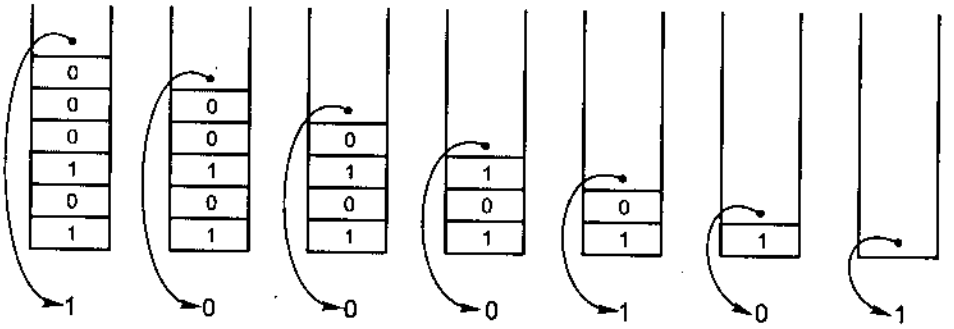
4.5. Các số sau đây sẽ lần lượt được in ra 7 ; 9 ; 7 ; 4 ; 2.

4.6. a)

– Tình trạng của stack khi thực hiện bước 2



–.Tình trạng của stack khi thực hiện bước 3.



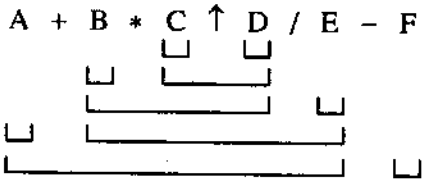
kết quả : $(69)_{10} = (1000101)_2$

4.7. Trước hết phải phân tích xem với mỗi toán tử (trong biểu thức trung tố) thì toán hạng 1 của nó là gì, toán hạng 2 của nó là gì.

Sau đó áp dụng quy tắc như đã nêu trong phần 1 mục 4.3.2.

Ví dụ :

c) Biểu thức A + B * C ↑ D / E – F sẽ được phân tích như sau :



Hai dấu ⌴ là ứng với 2 toán hạng của toán tử ở giữa chúng.

Từ đó có thể biến đổi sang dạng tiền tố như sau :

C ↑ D	sẽ chuyển thành	↑ C D
B * C ↑ D	–	* B ↑ C D
B * C ↑ D / E	–	/ * B ↑ C D E
A + B * C ↑ D / E	–	+ A / * B ↑ C D E

và cuối cùng biểu thức cho sẽ chuyển thành dạng tiền tố :

$$- + A / * B \uparrow C D E F -$$

Còn biến đổi sang dạng hậu tố thì sẽ như sau :

C ↑ D	sẽ chuyển thành	C D ↑
B * C ↑ D	–	B C D ↑ *
B * C ↑ D / E	–	B C D ↑ * E /
A + B * C ↑ D / E	–	A B C D ↑ * E / +

và cuối cùng biểu thức cho sẽ chuyển thành dạng hậu tố

A B C D ↑ * E / + F

Tương tự ta sẽ có kết quả đối với các biểu thức khác như sau :

	Tiền tố	hậu tố
a)	* + x y z	x y + z *
b)	– x – y * z + AB	x y z A B + * – –
d)	↑ * + A B D – E F	A B + D * E F – ↑

4.8. 1. Với biểu thức tiền tố ta sẽ đọc từ phải qua trái khi gặp một toán tử thi kết hợp nó với 2 toán hạng vừa gặp trước nó. Như biểu thức ở b), ta sẽ có :

D – E

B * C

A + (B * C)

(A + (B * C)) / (D – E)

a) Ta có biểu thức dạng trung tố sau :

(A + B) * (C – D)

2. Tương tự như câu 1, nhưng đọc biểu thức theo chiều từ trái qua phải. Ta sẽ có :

a) (A – (B + C)) * D

b) ((A + B) – C) / (D * E)

4.9. Làm tương tự như khi biến đổi sang dạng trung tố (như câu 2) bài tập 4.8) ; chỉ có khác là phép toán thực hiện ngay với giá trị của 2 toán hạng.

4.10.

Kí tự đọc	A	B	C	–	/	D	E	F	+	*	+
Tình trạng của stack								1.0			
			3.0				7.0	7.0	8.0		
		5.0	5.0	2.0		2.0	2.0	2.0	2.0	16.0	
	8.0	8.0	8.0	8.0	4.0	4.0	4.0	4.0	4.0	4.0	20.0

Chú ý : 1. Khi đọc tới toán tử, chẳng hạn phép trừ "–" thì 2 giá trị 3.0 và 5.0 lần lượt được lấy ra khỏi stack và 5.0 – 3.0 được thực hiện, kết quả bằng 2.0 sẽ được đẩy vào stack (đây là giá trị của một toán hạng ứng với phép chia "/" đọc sau đó).

2. Khi gặp dấu kết thúc biểu thức thì trong stack còn giá trị 20.0, đó chính là giá trị của biểu thức hậu tố đã cho.

4.11. Tình trạng của Q sau khi thực hiện các phép đã cho, sẽ như sau :

1	2	3	4	5	6
Oslo	Moscow				
F	R				

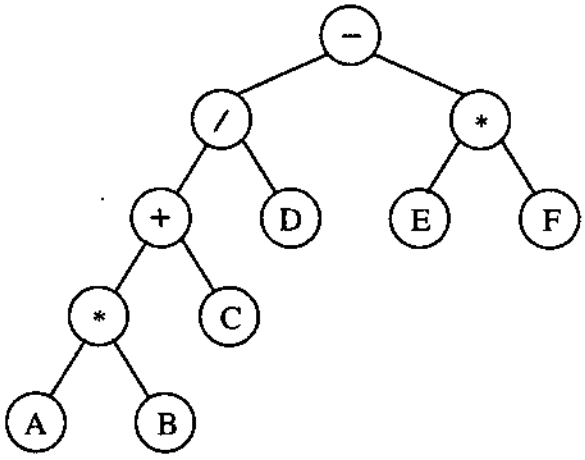
4.12. a) 5 phần tử ; b) 6 phần tử ; c) 2 phần tử.

CHƯƠNG 5

5.2. a) Vẽ một cây nhị phân suy biến với 7 nút.

b) Vẽ một cây nhị phân đầy đủ với 7 nút.

5.3. a) Cũng phải phân tích biểu thức để xác định 2 toán hạng, ứng với một toán tử (như bài tập 4.7 thuộc chương 4) sau đó áp dụng quy tắc như đã nêu ở mục 5.2.1. thuộc chương 5. Cuối cùng ta sẽ có cây nhị phân :



5.4. a) R S X Y Z T U W P V.

b) X S Z Y R W P U T V.

c) X Z Y S P W U V T R.

5.5. a) Chú ý là 2x phải hiểu là 2 * x

5a phải hiểu là 5 * a

b) Muốn có biểu thức dạng tiền tố thì phải duyệt cây T theo thứ tự trước, dạng hậu tố thì phải duyệt cây T theo thứ tự sau.

5.6. Nếu gọi n(T) là hàm có giá trị bằng số nút của một cây nhị phân T, thì n(T) có thể được định nghĩa như sau :

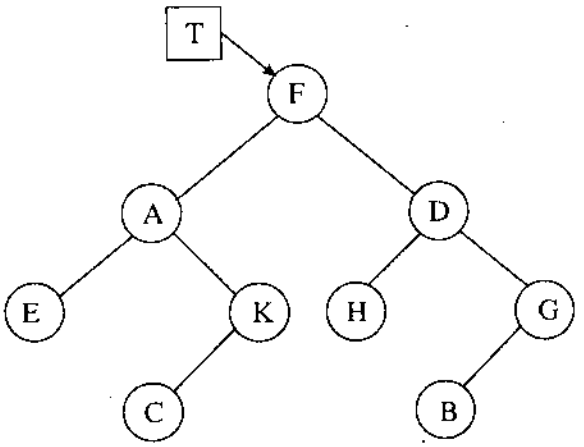
$$n(T) = \begin{cases} 0 & \text{nếu T = null (cây rỗng)} \\ 1 + n(LPTR(T)) + n(RPTR(T)) & \text{nếu T} \neq \text{null} \end{cases}$$

Đây chính là định nghĩa đệ quy của n(T).

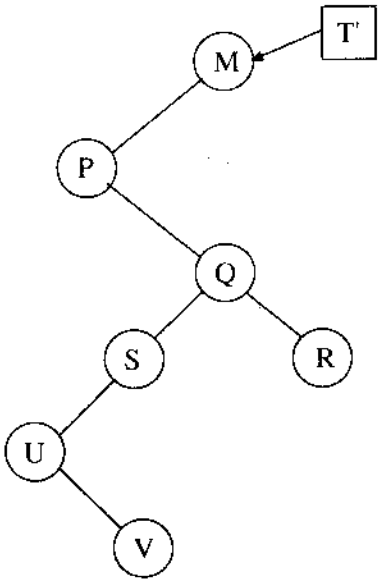
Nếu dựa vào định nghĩa này có thể viết được giải thuật đệ quy để tính n(T).

5.7. Dựa vào thứ tự bước ta có thể xác định được nút gốc cây. Nếu đã biết nút gốc thì dựa vào thứ tự giữa ta sẽ xác định được các nút của cây con trái và các nút của cây con phải. Với cây con trái hoặc cây con phải này, việc xác định gốc và các nút thuộc hai cây con của gốc này lại được thực hiện tương tự.

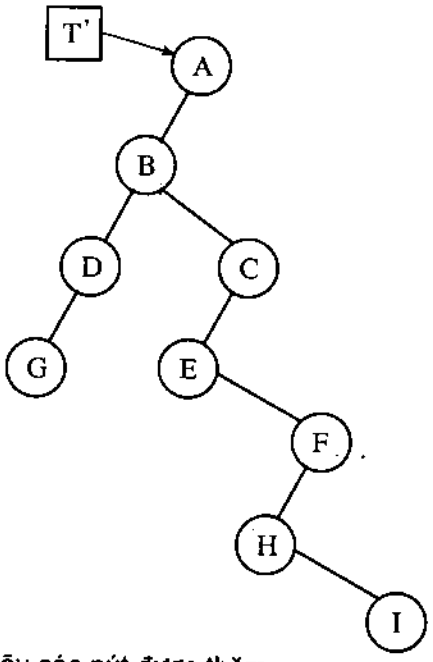
Cuối cùng ta sẽ dựng được cây T như sau :



5.8. Cây nhị phân tương đương T' ứng với cây hình 5.37 có dạng :



5.9. a) Cây T' sẽ có dạng như sau :



b) Bảng so sánh dãy các nút được thăm

Duyệt theo thứ tự	Cây T	Cây T'
Thứ tự trước	A B D G C E F H I	A B D G C E F H I
Thứ tự giữa	D G B A E C H F I	G D B E H I F C A
Thứ tự sau	G D B E H I F C A	G D I H F E C B A

Ta thấy :

• Theo thứ tự trước thì phép duyệt cây T cho kết quả giống như phép duyệt cây T'

• Theo thứ tự sau thì phép duyệt cây T cho kết quả lại giống như kết quả của phép duyệt cây T' theo thứ tự giữa.

• Còn kết quả của phép duyệt cây T theo thứ tự giữa không giống bất kì kết quả của phép duyệt nào đối với cây T' cả.

c) Vì cây T' là cấu trúc lưu trữ của cây T, các xử lí đều thực hiện trên T'.

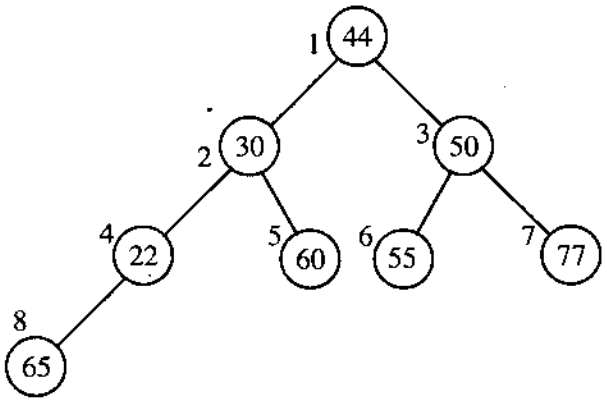
Phép duyệt theo thứ tự trước của cây T sẽ được thực hiện bằng phép duyệt theo thứ tự trước của cây T'.

Phép duyệt theo thứ tự sau của cây T sẽ được thực hiện bằng phép duyệt theo thứ tự giữa của cây T' ;

Còn phép duyệt theo thứ tự giữa của T không thể thực hiện được trong máy với cây T', nên không đặt ra.

Có như vậy mới giữ được sự nhất quán giữa định nghĩa phép duyệt cây tổng quát với việc thực hiện nó trong máy, qua cây T'.

5.11. a) Cây đó có dạng :



b) Hãy đối chiếu với định nghĩa của đống để thấy cây này không thỏa mãn điều kiện về số gắn ở các nút trên cây.

c) Muốn tạo thành đống cho cây này thì phải thực hiện câu lệnh :

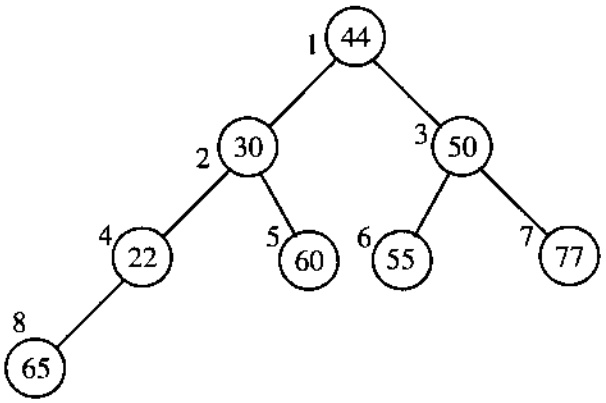
for i := [n/2] **down to** 1 **do**

call ADJUST (i, n) ;

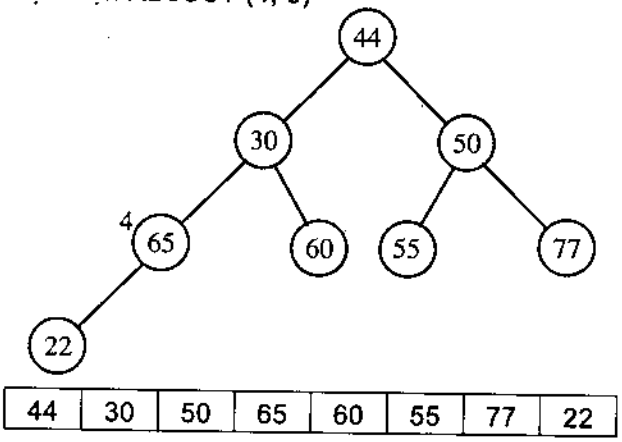
trong giải thuật HEAP – SORT

Sau đây là diễn biến của dạng cây và cấu trúc lưu trữ tương ứng.

1. Dạng cây lúc ban đầu :



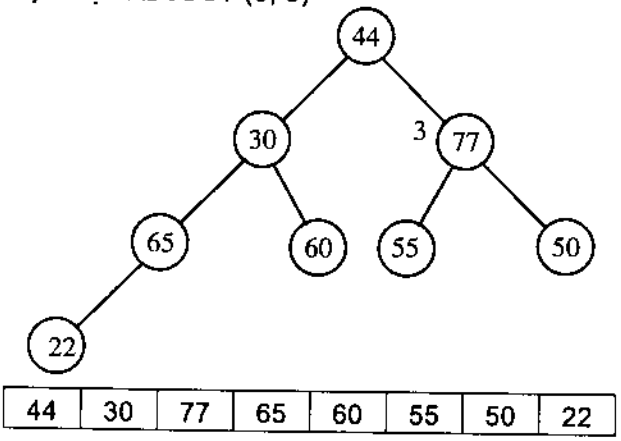
2. Sau khi thực hiện ADJUST (4, 8)



44	30	50	65	60	55	77	22

Vec tơ lưu trữ cây

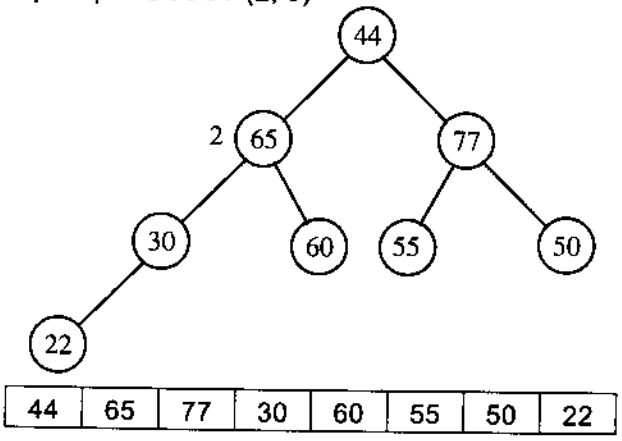
3. Sau khi thực hiện ADJUST (3, 8)



44	30	77	65	60	55	50	22

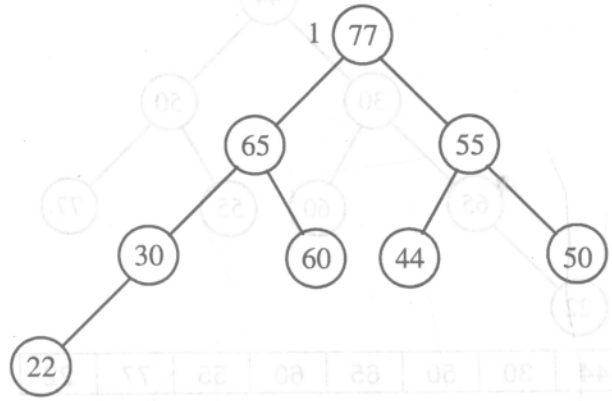
vectơ lưu trữ cây

4. Sau khi thực hiện ADJUST (2, 8)



44	65	77	30	60	55	50	22

5. Sau khi thực hiện ADJUST (1,8) : cây đã là đống



77	65	55	30	60	44	50	22

d) Nếu dựa vào đống này để tiến hành sắp xếp thì phải thực hiện tiếp câu lệnh :

"**for** i := n – 1 **down to** 1 **do begin**

A[1] ↔ A[i + 1] ; {đổi chỗ}

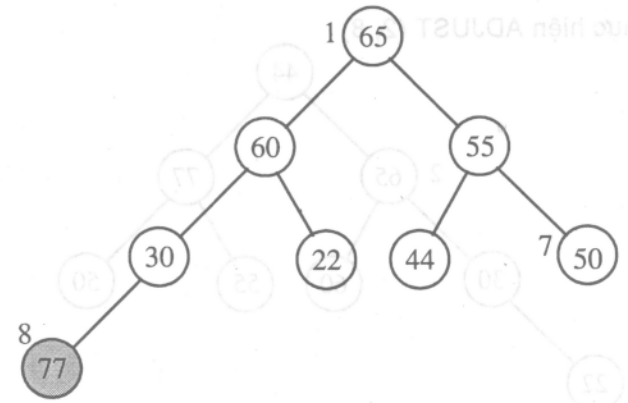
call ADJUST (1, i) {vun đống}

end

trong giải thuật HEAP – SORT.

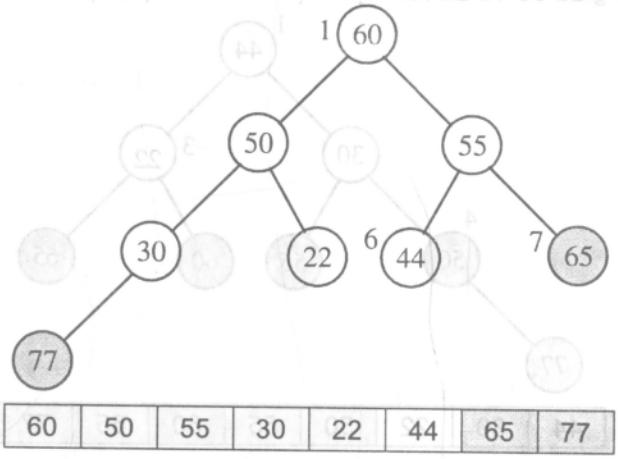
Sau đây là diễn biến của dạng cây và cấu trúc lưu trữ tương ứng :

1. Đổi chỗ giữa 77 và 22 rồi thực hiện ADJUST (1, 7)



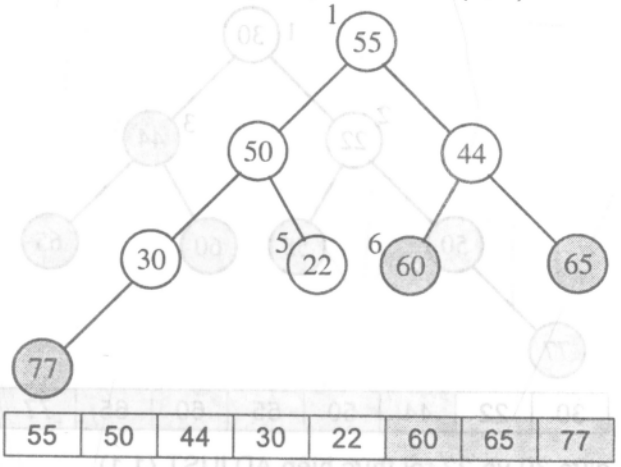
65	60	55	30	22	44	50	77

2. Đổi chỗ giữa 65 và 50 rồi thực hiện ADJUST (1, 6)



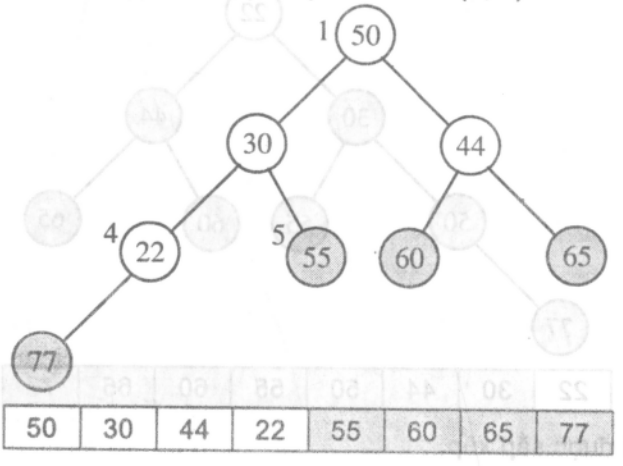
60	50	55	30	22	44	65	77

3. Đổi chỗ giữa 60 và 44 rồi thực hiện ADJUST (1, 5)



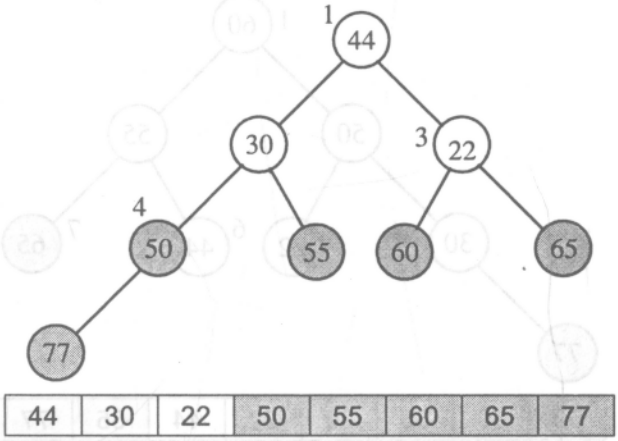
55	50	44	30	22	60	65	77

4. Đổi chỗ giữa 55 và 22 rồi thực hiện ADJUST (1, 4)

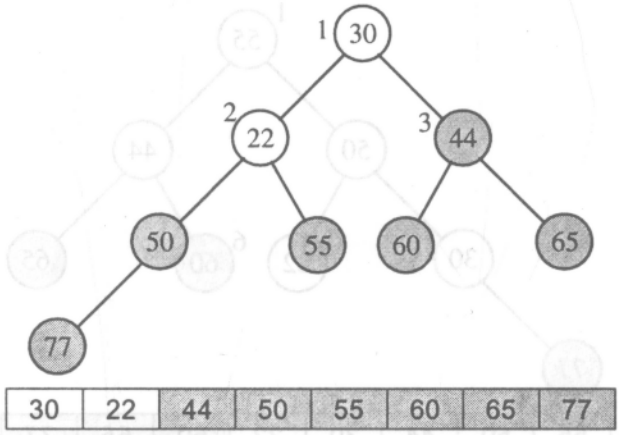


50	30	44	22	55	60	65	77

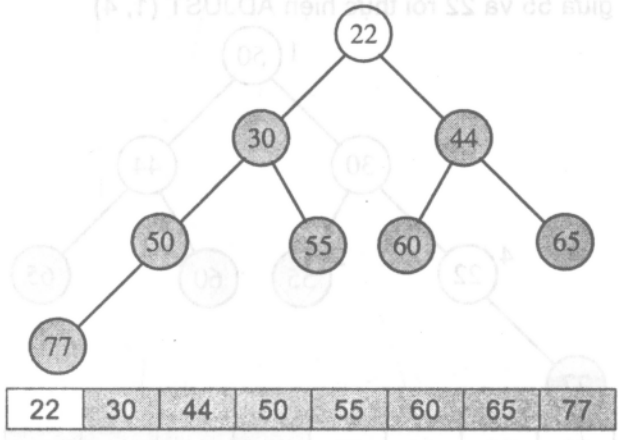
5. Đổi chỗ giữa 50 và 22 rồi thực hiện ADJUST (1, 3)



6. Đổi chỗ giữa 44 và 22 rồi thực hiện ADJUST (1, 2)



7. Đổi chỗ giữa 30 và 22 rồi thực hiện ADJUST (1,1)

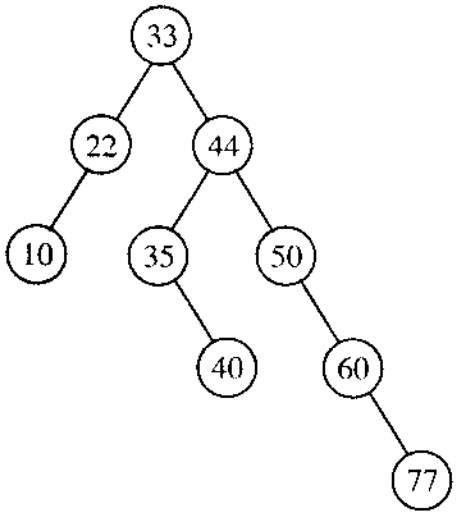


Dãy số đã được sắp xếp.

5.12. Khái niệm "đống" phải thay đổi ở chỗ : "số ở nút cha bao giờ cũng nhỏ hơn số ở nút con". Như vậy khi đã có một đống thì số nhỏ nhất chứa ở nút gốc.

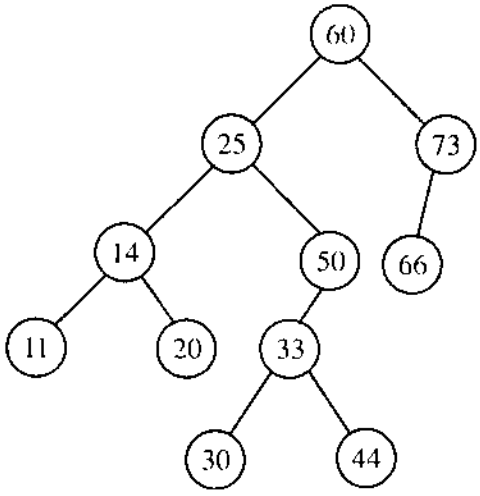
Tất nhiên giải thuật ADJUST (i, n) để chỉnh lí thành đống cho một cây nhị phân với gốc có số thứ tự là i và 2 con của gốc đã là đống rồi, cũng phải thay đổi theo.

5.13.

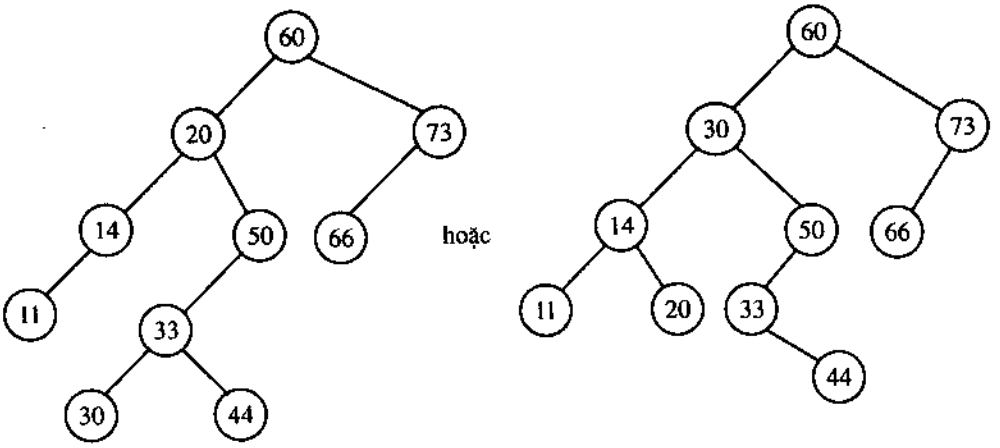


5.14. b) Xem lại ví dụ ở mục 5.62 (ứng với hình 5.30) để phỏng theo.

5.16. a)



b)



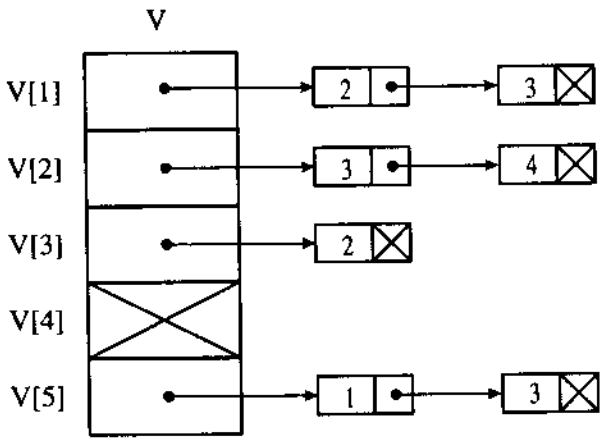
CHƯƠNG 6

6.3. Nếu đồ thị vô hướng có n đỉnh thì có tối đa $\frac{n(n-1)}{2}$ cung, nối một đỉnh với một đỉnh khác (có thể chứng minh được bằng quy nạp toán học).

6.5. a) Ma trận lân cận của G_2 có dạng :

$$A = \begin{bmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \end{bmatrix}$$

b) Danh sách lân cận của G_1 có dạng :



6.6.

$$A = \begin{bmatrix} 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 \end{bmatrix}$$

$$A^{(2)} = \begin{bmatrix} 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix}$$

$$A^{(3)} = \begin{bmatrix} 1 & 1 & 0 & 1 \\ 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 \end{bmatrix}$$

$$A^{(4)} = \begin{bmatrix} 1 & 1 & 0 & 1 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 0 & 0 \end{bmatrix}$$

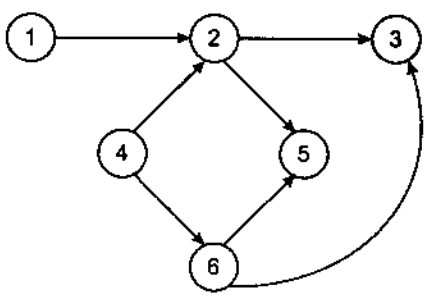
$$P = \begin{bmatrix} 1 & 1 & 0 & 1 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 0 & 1 \end{bmatrix}$$

Ta thấy : P[3, 1] = 1, chứng tỏ : có đường đi từ đỉnh 3 tới đỉnh 1. Đó chính là đường đi 3, 2, 1 hoặc 3, 4, 2, 1 trên G_3.

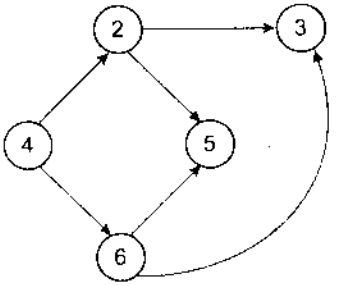
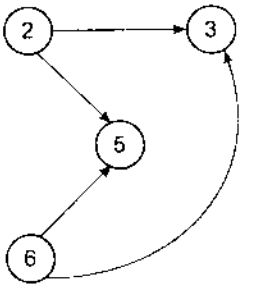
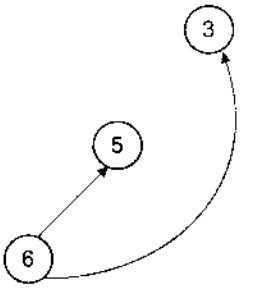
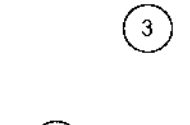
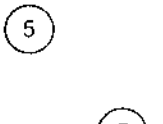
P[1, 3] = P[2, 3] = P[3, 3] = P[4, 3] = 0, chứng tỏ : không một đường đi nào xuất phát từ một đỉnh khác tới đỉnh 3 cả.

P[2, 2] = 1, chứng tỏ : có chu trình đi từ đỉnh 2 rồi quay về đỉnh 2. Đó chính là đường đi 2, 1, 2 hoặc 2, 1, 4, 2.

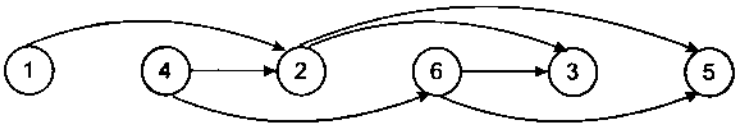
6.8. a)



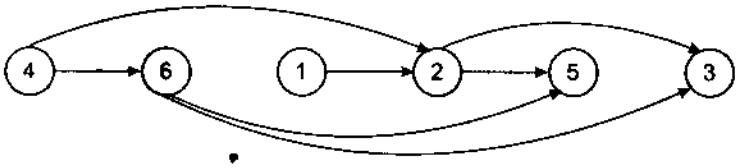
b) Thứ tự tôpô sẽ như sau :

Đỉnh đưa ra sắp xếp	Phần đồ thị còn lại
1	
4	
2	
6	
3	
5	

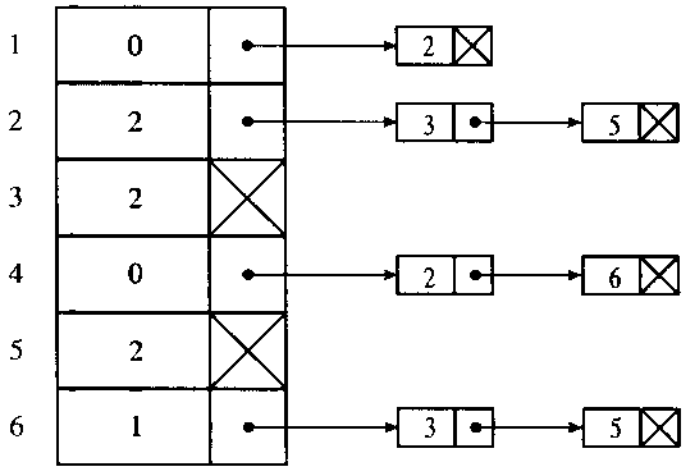
cuối cùng ta sẽ có thứ tự tôpô minh họa qua đồ thị như sau :



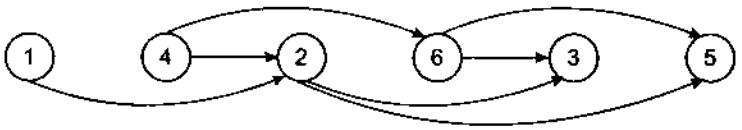
Bằng cách tương tự với đỉnh đưa ra đầu tiên là (4), ta sẽ có được thứ tự tô pô khác :



c)



d)



6.9. Cách làm tương tự như bài 6.8.

TÀI LIỆU THAM KHẢO

[1] ĐỖ XUÂN LÔI, "Cấu trúc dữ liệu và giải thuật", Nhà xuất bản Giáo dục, 1993.

[2] SEYMOUR LIPSCHUTZ, "Les Structures de données" McGraw – Hill, 1987.

[3] ELLIS HOROWITZ, SARTAJ SAHNI, "Fundamental of data structures", Computer science press INC, 1976.

[4] MICHAEL T.GOODRICH, ROBERTO TAMASSIA, "Data Structures and Algorithms in JAVA", 1998.

MỤC LỤC

Chương 5

CẤU TRÚC CÂY (tree)

Chương 6

ĐỒ THỊ (graph)

Chịu trách nhiệm xuất bản :

Chủ tịch HĐQT kiêm Tổng Giám đốc NGÔ TRẦN ÁI

Phó Tổng Giám đốc kiêm Tổng biên tập NGUYỄN QUÝ THAO

Biên tập nội dung :

TRẦN VĂN THẮNG

Trình bày bìa :

TÀO HUYỀN

Sửa bản in :

TRẦN VĂN THẮNG

Chế bản :

MINH HUYỀN

GIÁO TRÌNH CẤU TRÚC DỮ LIỆU VÀ GIẢI THUẬT

Mã số: 6H160T6 - DAI

In 1.500 bản, khổ 16 x 24 cm, tại Công ty CP In Phúc Yên.

Số xuất bản: 04 - 2006/CXB/33-1860/GD

In xong và nộp lưu chiểu quý III năm 2006.

CÔNG TY CỔ PHẦN SÁCH ĐẠI HỌC - DẠY NGHỀ
HEVOBCO
Địa chỉ : 25 Hàn Thuyên, Hà Nội

INTERNATIONAL QUALITY AWARD
PLATINUM
2004
NGÔI SAO BẠCH KIM
CHẤT LƯỢNG
QUỐC TẾ

TÌM ĐỌC GIÁO TRÌNH DÙNG CHO CÁC TRƯỜNG ĐÀO TẠO HỆ TRUNG HỌC CHUYÊN NGHIỆP - DẠY NGHỀ CỦA NHÀ XUẤT BẢN GIÁO DỤC

(NGÀNH ĐIỆN TỬ - TIN HỌC)

1.	**Linh kiện điện tử và ứng dụng**	*TS. Nguyễn Viết Nguyên*
2.	**Điện tử dân dụng**	*ThS. Nguyễn Thanh Trà*
3.	**Điện tử công suất**	*Trần Trọng Minh*
4.	**Mạch điện tử**	*TS. Đặng Văn Chuyết*
5.	**Kĩ thuật số**	*TS. Nguyễn Viết Nguyên*
7.	**Kĩ thuật điều khiển**	*.Vũ Quang Hồi*
8.	**Kỹ thuật xung - số**	*TS. Lương Ngọc Hải*
9.	**Điện tử công nghiệp**	*Vũ Quang Hồi*
10.	**Toán ứng dụng trong tin học**	*PGS. TS. Bùi Minh Trí*
11.	**Nhập môn tin học**	*Tô Văn Nam*
12.	**Cấu trúc máy vi tính và vi xử lý**	*Lê Hải Sâm - Phạm Thanh Liêm*
13.	**Hệ các chương trình ứng dụng (Window, Word, Excel)**	*GVC. Trần Viết Thường - Tô Văn Nam*
14.	**Cơ sở dữ liệu**	*Tô Văn Nam*
15.	**Lập trình C**	*GVC Tiêu Kim Cương*
16.	**Cấu trúc dữ liệu và giải thuật**	*PGS.TS. Đỗ Xuân Lôi*
17.	**Cài đặt và điều hành mạng**	*TS. Nguyễn Vũ Sơn*
18.	**Phân tích thiết kế hệ thống**	*GVC. Tô Văn Nam*
19.	**ACCESS và ứng dụng**	*TS. Huỳnh Quyết Thắng*
20.	**Sử dụng Corel Draw**	*Nguyễn Phú Quảng*
21.	**Bảo trì và quản lí phòng máy tính**	*Phạm Thanh Liêm*
22.	**Kinh tế và quản trị doanh nghiệp (kinh tế và TCQLSX)**	*TS. Ngô Xuân Bình - TS. Hoàng Văn Hải*

Bạn đọc có thể tìm mua tại các Công ti Sách - Thiết bị trường học ở các địa phương hoặc các Cửa hàng sách của Nhà xuất bản Giáo dục:

Tại Hà Nội : ***25 Hàn Thuyên, 81 Trần Hưng Đạo, 187 Giảng Võ, 23 Tràng Tiền.***
Tại Đà Nẵng : ***15 Nguyễn Chí Thanh.***
Tại Thành phố Hồ Chí Minh : ***104 Mai Thị Lựu, Quận 1.***

8 934980 522486

Giá: 14.000đ