HỌC VIỆN CÔNG NGHỆ BƯU CHÍNH VIỄN THÔNG

Giáo trình

KỸ THUẬT LẬP TRÌNH

NHÀ XUẤT BẢN BƯU ĐIỆN

Giáo trình

KỸ THUẬT LẬP TRÌNH

HỌC VIỆN CÔNG NGHỆ BƯU CHÍNH VIỄN THÔNG

Giáo trình

KỸ THUẬT LẬP TRÌNH

BIÊN SOẠN:

TS. Lê Hữu Lập
ThS. Nguyễn Duy Phương

NHÀ XUẤT BẢN BƯU ĐIỆN

Hà Nội, tháng 12 - 2002

Mã số: GT 14 HM 06

MỤC LỤC

LỜI GIỚI THIỆU

Sự phát triển công nghệ thông tin trong những năm vừa qua đã làm thay đổi bộ mặt kinh tế xã hội toàn cầu, trong đó công nghệ phần mềm trở thành một ngành công nghiệp quan trọng và đầy tiềm năng. Hiện nay trong các trường đào tạo đại học về kỹ thuật và công nghệ đều hướng cho sinh viên sau khi ra trường có các kỹ năng về lập trình trên máy tính. Ngoài việc tạo lập cho họ điều kiện nghiên cứu để làm chủ các hệ thống phần mềm, kỹ năng lập trình còn giúp họ xây dựng và phát triển mới các sản phẩm phần mềm chuyên dụng.

Với sự hội tụ của công nghệ viễn thông và công nghệ thông tin, tỷ trọng về giá trị phần mềm chiếm rất cao trong các hệ thống viễn thông cũng như các thiết bị đầu cuối. Chính vì lý do đó, việc nghiên cứu, tìm hiểu, làm chủ và tiến tới phát triển được các hệ thống phần mềm của các kỹ sư công nghệ thông tin và kỹ sư điện tử viễn thông là rất cần thiết. Tuy nhiên, chúng ta phải bắt đầu từ các kỹ thuật lập trình cơ bản mà hiện thân của nó là kỹ thuật lập trình có cấu trúc.

*Nắm bắt được nhu cầu trên, Học viện Công nghệ Bưu chính Viễn thông phối hợp với Nhà xuất bản Bưu điện xuất bản cuốn giáo trình "**Kỹ thuật lập trình**" do TS. Lê Hữu Lập và ThS. Nguyễn Duy Phương biên soạn. Cuốn giáo trình này sẽ giới thiệu với bạn đọc về các kỹ năng lập trình cấu trúc thông qua các thuật toán quan trọng. Với nhiều ví dụ cụ thể, cuốn sách giúp cho bạn đọc thuận tiện trong quá trình học tập nghiên cứu để tạo cho mình kỹ năng lập trình cơ bản. Đây là giáo trình phục vụ đào tạo tại Học viện Công nghệ Bưu chính Viễn thông.*

Chúng tôi hy vọng cuốn sách sẽ là tài liệu tham khảo hữu ích cho giảng viên, sinh viên các trường đại học về kỹ thuật và công nghệ có môn học "Kỹ thuật lập trình"..

Xin trân trọng giới thiệu.

Hà Nội, ngày 12 tháng 12 năm 2002

HỌC VIỆN CÔNG NGHỆ BƯU CHÍNH VIỄN THÔNG

LỜI NÓI ĐẦU

*Môn học **Kỹ thuật lập trình** là môn cơ sở bắt buộc đối với sinh viên chuyên ngành điện tử viễn thông và công nghệ thông tin của Học viện công nghệ Bưu chính Viễn thông.*

Trên cơ sở các kinh nghiệm đã được đúc rút từ bài giảng của môn học Kỹ thuật lập trình cho sinh viên các ngành nói trên trong những năm học vừa qua, với mục đích cung cấp cho sinh viên những kiến thức cơ bản nhất, có tính hệ thống liên quan tới môn học, chúng tôi đã mạnh dạn xây dựng cuốn giáo trình này.

Khi học môn Kỹ thuật lập trình, sinh viên chỉ cần học qua môn "Tin học cơ bản" và chỉ cần thế các bạn đã có đủ kiến thức cơ sở cần thiết để tiếp thu kiến thức của Kỹ thuật lập trình.

Thông qua cuốn giáo trình này, chúng tôi muốn giới thiệu với các bạn đọc về kỹ năng lập trình cấu trúc thông qua một số thuật toán quan trọng, bao gồm: Đại cương về lập trình cấu trúc; Con trỏ và mảng; Duyệt và đệ qui; Ngăn xếp, hàng đợi và danh sách móc nối; Cây; Đồ thị và cuối cùng là Sắp xếp và tìm kiếm. Phần phụ lục là bài tập tổng hợp lại những kiến thức cơ bản nhất đã được đề cập trong giáo trình và được thể hiện bằng một chương trình.

Tuy đã rất chú ý và cẩn trọng trong quá trình biên soạn, nhưng giáo trình chắc chắn không tránh khỏi những thiếu sót và hạn chế. Chúng tôi xin chân thành mong bạn đọc đóng góp ý kiến để giáo trình nay ngày càng hoàn thiện hơn. Mọi sự đóng góp ý kiến xin gửi về Khoa Công nghệ thông tin - Học viện Công nghệ Bưu chính Viễn thông.

Chúng tôi xin tỏ lòng biết ơn tới TS. Từ Minh Phương, giảng viên khoa Công nghệ thông tin - Học viện Công nghệ Bưu chính Viễn thông đã đọc và hiệu đính lại toàn bộ bản thảo của giáo trình này.

Hà Nội, ngày 24 tháng 8 năm 2002

CÁC TÁC GIẢ

CHƯƠNG 1

ĐẠI CƯƠNG VỀ KỸ THUẬT LẬP TRÌNH CẤU TRÚC

1.1. SƠ LƯỢC VỀ LỊCH SỬ LẬP TRÌNH CẤU TRÚC

Lập trình là một trong những công việc nặng nhọc nhất của khoa học máy tính. Có thể nói, năng suất xây dựng các sản phẩm phần mềm là rất thấp so với các hoạt động trí tuệ khác. Cho đến nay, năng suất của một lập trình viên chỉ dao động trong khoảng 3-5 lệnh/ngày. Một sản phẩm phần mềm có thể được thiết kế và cài đặt trong vòng 6 tháng với 3 lao động chính. Nhưng để kiểm tra tìm lỗi và tiếp tục hoàn thiện sản phẩm đó phải mất thêm chừng 3 năm. Đây là hiện tượng phổ biến trong tin học của những năm 1960 khi xây dựng các sản phẩm phần mềm bằng kỹ thuật lập trình tuyến tính. Để khắc phục tình trạng lỗi của sản phẩm, người ta che chắn nó bởi một mành che mang tính chất thương mại được gọi là version (phiên bản). Thực chất, version là việc thay thế sản phẩm cũ bằng cách sửa đổi nó rồi công bố dưới dạng một version mới, giống như: MS-DOS 4.0 chỉ tồn tại trong thời gian vài tháng rồi thay đổi thành MS-DOS 5.0, MS-DOS 5.5, MS-DOS 6.0... Đây không phải là một sản phẩm mới như ta tưởng mà trong nó còn tồn tại những lỗi không thể bỏ qua được, vì ngay MS-DOS 6.0 cũng chỉ là sự khắc phục hạn chế của MS-DOS 3.3 ban đầu.

Trong thời kỳ đầu của tin học, các lập trình viên xây dựng chương trình bằng các ngôn ngữ lập trình bậc thấp, quá trình nạp và theo dõi hoạt động của chương trình một cách trực tiếp trong chế độ trực tuyến (on-line). Việc tìm và sửa lỗi (debugging) như ngày nay là không thể thực hiện được. Do vậy, trước những năm 1960, người ta coi việc lập trình giống như những hoạt động nghệ thuật nhuộm màu sắc cá nhân hơn là khoa học. Một số người nắm được một vài ngôn ngữ lập trình,

cùng một số mẹo vặt tận dụng cấu hình vật lý cụ thể của hệ thống máy tính, tạo nên một số món lạ của phần mềm được coi là một chuyên gia nắm bắt được những bí ẩn của nghệ thuật lập trình.

Các hệ thống máy tính trong giai đoạn này có cấu hình yếu, bộ nhớ nhỏ, tốc độ các thiết bị vào ra thấp làm chậm quá trình nạp và thực hiện chương trình. Chương trình được xây dựng bằng kỹ thuật lập trình tuyến tính mà nổi bật nhất là ngôn ngữ lập trình Assembler và Fortran. Với phương pháp lập trình tuyến tính, lập trình viên chỉ được phép thể hiện chương trình của mình trên hai cấu trúc lệnh, đó là cấu trúc lệnh tuần tự (sequential) và nhảy không điều kiện (goto). Hệ thống thư viện vào/ra nghèo nàn làm cho việc lập trình trở nên khó khăn, chi phí cho các sản phẩm phần mềm quá lớn, độ tin cậy của các sản phẩm phần mềm không cao dẫn tới hàng loạt các dự án tin học bị thất bại, đặc biệt là các hệ thống tin học có tầm cỡ lớn. Năm 1973, Hoare khẳng định, nguyên nhân thất bại mà người Mỹ gặp phải khi phóng vệ tinh nhân tạo về phía sao Vệ nữ (Sao Kim) là do lỗi của chương trình điều khiển viết bằng Fortran. Thay vì viết:

```
DO 50 I = 12, 523
```

(Thực hiện số 50 với I là 12, 13, ..., 523)

Lập trình viên (hoặc thao tác viên đục bìa) viết thành:

```
DO 50 I = 12.523
```

(Dấu phẩy đã thay bằng dấu chấm)

Gặp câu lệnh này, chương trình dịch của Fortran đã hiểu là gán giá trị thực 12.523 cho biến DO 50 I làm cho kết quả chương trình sai.

Để giải quyết những vướng mắc trong kỹ thuật lập trình, các nhà tin học lý thuyết đã đi sâu vào nghiên cứu tìm hiểu bản chất của ngôn ngữ, thuật toán và hoạt động lập trình, nâng nội dung của kỹ thuật lập trình lên thành các nguyên lý khoa học như ngày nay. Kết quả nổi bật nhất trong giai đoạn này là Knuth xuất bản bộ 3 tập sách mang tên "Nghệ thuật lập trình" giới thiệu hết sức tỉ mỉ cơ sở lý thuyết đảm bảo toán học

và các thuật toán cơ bản xử lý dữ liệu nửa số, sắp xếp và tìm kiếm. Năm 1968, Dijkstra công bố lá thư "Về sự nguy hại của toán tử GOTO". Trong công trình này, Dijkstra khẳng định, có một số lỗi do GOTO gây nên không thể xác định được điểm bắt đầu của lỗi. Dijkstra còn khẳng định thêm: "Tay nghề của một lập trình viên tỉ lệ nghịch với số lượng toán tử GOTO mà anh ta sử dụng trong chương trình", đồng thời kêu gọi huỷ bỏ triệt để toán tử GOTO trong mọi ngôn ngữ lập trình ngoại trừ ngôn ngữ lập trình bậc thấp. Dijkstra còn đưa ra khẳng định, động thái của chương trình có thể được đánh giá tường minh qua các cấu trúc lặp, rẽ nhánh, gọi đệ qui là cơ sở của lập trình cấu trúc ngày nay.

Những kết quả được Dijkstra công bố đã tạo nên một cuộc cách mạng trong kỹ thuật lập trình, Knuth liệt kê một số trường hợp có lợi của GOTO như vòng lặp kết thúc giữa chừng, bắt lỗi..., Dijkstra, Hoare, Knuth tiếp tục phát triển tư tưởng coi chương trình máy tính và lập trình viên là đối tượng nghiên cứu của kỹ thuật lập trình và phương pháp làm chủ sự phức tạp của các hoạt động lập trình. Năm 1969, Hoare đã phát biểu các tiên đề phục vụ cho việc chứng minh tính đúng đắn của chương trình, phát hiện tính bất biến của vòng lặp bằng cách coi chương trình vừa là bản mã hoá thuật toán đồng thời là bản chứng minh tính đúng đắn của chương trình. Sau đó Dahl, Hoare, Dijksta đã phát triển thành ngôn ngữ lập trình cấu trúc.

Để triển khai các nguyên lý lập trình cấu trúc, L. Wirth đã thiết kế và cài đặt ngôn ngữ ALGOL W là một biến thể của ALGOL 60. Sau này, L. Wirth tiếp tục hoàn thiện để trở thành ngôn ngữ lập trình Pascal. Đây là ngôn ngữ lập trình giản dị, sáng sủa về cú pháp, dễ minh họa những vấn đề phức tạp của lập trình hiện đại và được coi là một chuẩn mực trong giảng dạy lập trình.

Năm 1978, Brian Barninghan cùng Denit Ritche thiết kế ngôn ngữ lập trình C với tối thiểu các cấu trúc lệnh và hàm, khá phù hợp với tư duy và tâm lý của của người lập trình. Đồng thời, hai tác giả đã phát hành phiên bản hệ điều hành UNIX viết chủ yếu bằng ngôn ngữ C, khẳng định thêm uy thế của C trong lập trình hệ thống.

1.2. CẤU TRÚC LỆNH, LỆNH CÓ CẤU TRÚC, CẤU TRÚC DỮ LIỆU

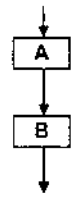
1.2.1. Cấu trúc lệnh (cấu trúc điều khiển)

Mỗi chương trình máy tính về bản chất là một bản mã hoá thuật toán. Thuật toán được coi là dãy hữu hạn các thao tác sơ cấp trên tập đối tượng vào (Input) nhằm thu được kết quả ra (Output). Các thao tác trong một ngôn ngữ lập trình cụ thể được điều khiển bởi các lệnh hay các cấu trúc điều khiển, còn các đối tượng chịu thao tác thì được mô tả và biểu diễn thông qua các cấu trúc dữ liệu.

Cấu trúc tuần tự:

A;

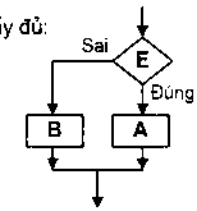
B;



Sau khi thực hiện lệnh A thì thực hiện lệnh B

Cấu trúc rẽ nhánh dạng đầy đủ:

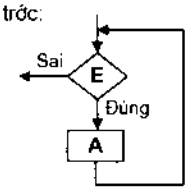
If (E) **A**;
Else **B**;



Nếu biểu thức E có giá trị đúng (khác 0) thì thực hiện A; Nếu E sai thì thực hiện B;

Cấu trúc lặp với điều kiện trước:

While (E) **A**;

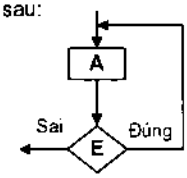


Trong khi biểu thức E còn có giá trị đúng thì thực hiện A;

Cấu trúc lặp với điều kiện sau:

Do

A;
While (E);

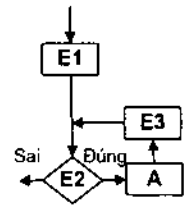


Thực hiện A cho tới khi nào E vẫn còn đúng;

Cấu trúc lặp FOR:

For (**E1**; **E2**; **E3**)

A;



Thực hiện E1; kiểm tra E2 nếu E2 có giá trị đúng thì thực hiện A; quá trình đợc lặp lại bằng việc thực hiện E3 và kiểm tra E2;

Trong các ngôn ngữ lập trình cấu trúc, những cấu trúc lệnh sau được sử dụng để xây dựng chương trình. Dĩ nhiên, chúng ta sẽ không bàn tới cấu trúc nhảy không điều kiện GOTO mặc dù ngôn ngữ lập trình cấu trúc nào cũng trang bị cấu trúc lệnh GOTO.

A, B: ký hiệu cho các câu lệnh đơn hoặc lệnh hợp thành. Mỗi lệnh đơn lẻ được gọi là một lệnh đơn, lệnh hợp thành là lệnh hay cấu trúc lệnh được ghép lại với nhau theo qui định của ngôn ngữ, trong Pascal là tập lệnh hay cấu trúc lệnh được bao trong thân của Begin ... End; trong C là tập các lệnh hay cấu trúc lệnh được bao trong hai ký hiệu { ... }.

E, E1, E2, E3 là các biểu thức số học hoặc logic. Một số ngôn ngữ lập trình coi giá trị của biểu thức logic hoặc đúng (TRUE) hoặc sai (FALSE), một số ngôn ngữ lập trình khác như C coi giá trị của biểu thức logic là đúng nếu nó có giá trị khác 0, ngược lại biểu thức logic có giá trị sai.

Cần lưu ý rằng, một chương trình được thể hiện bằng các cấu trúc điều khiển lệnh: tuần tự, tuyển chọn If ... Else, Switch ... Case ... Default, lặp với điều kiện trước While, lặp với điều kiện sau Do ... While, vòng lặp FOR bao giờ cũng chuyển được về một chương trình, chỉ sử dụng tối thiểu hai cấu trúc lệnh là tuần tự và lặp với điều kiện trước While. Phương pháp lập trình này còn được gọi là phương pháp lập trình hạn chế.

1.2.2. Lệnh có cấu trúc

Lệnh có cấu trúc là lệnh cho phép chứa các cấu trúc điều khiển trong nó. Khi tìm hiểu một cấu trúc điều khiển cần xác định rõ vị trí được phép đặt một cấu trúc điều khiển trong nó, cũng như nó là một phần của cấu trúc điều khiển nào. Điều này tưởng như rất tầm thường nhưng có ý nghĩa hết sức quan trọng trong khi xây dựng và kiểm tra lỗi có thể xảy ra trong chương trình. Nguyên tắc viết chương trình theo cấu trúc: Cấu trúc con phải được viết lọt trong cấu trúc cha, điểm vào và điểm ra của mỗi cấu trúc phải nằm trên cùng một hàng dọc. Ví dụ sau sẽ minh họa cho nguyên tắc viết chương trình:

```
if(E)
        while(E1)
                A;
        else
do
                B;
        while(E2);
```

Trong ví dụ trên, while(E1) A; là cấu trúc con nằm trong thân của cấu trúc cha là if(E); còn do B while(E2) là cấu trúc con trong thân của else. Do vậy, câu lệnh while(E1); do ... while(E2) có cùng cấp với nhau nên nó phải nằm trên cùng một cột, tương tự như vậy với A, B và if với else.

1.2.3. Cấu trúc dữ liệu

Các ngôn ngữ lập trình cấu trúc nói chung đều giống nhau về cấu trúc lệnh và cấu trúc dữ liệu. Điểm khác nhau duy nhất giữa các ngôn ngữ lập trình cấu trúc là phương pháp đặt tên, cách khai báo, cú pháp câu lệnh và tập các phép toán được phép thực hiện trên các cấu trúc dữ liệu cụ thể. Nắm bắt được nguyên tắc này, chúng ta sẽ dễ dàng chuyển đổi cách thể hiện chương trình từ ngôn ngữ lập trình này sang ngôn ngữ lập trình khác một cách nhanh chóng mà không tốn quá nhiều thời gian cho việc học tập ngôn ngữ lập trình.

Thông thường, các cấu trúc dữ liệu được phân thành hai loại: cấu trúc dữ liệu có kiểu cơ bản (Base type) và cấu trúc dữ liệu có kiểu do người dùng định nghĩa (User type) hay còn gọi là kiểu dữ liệu có cấu trúc. Kiểu dữ liệu cơ bản bao gồm: Kiểu ký tự (char), kiểu số nguyên có dấu (signed int), kiểu số nguyên không dấu (unsigned int), kiểu số nguyên dài có dấu (signed long), kiểu số nguyên dài không dấu (unsigned long), kiểu số thực (float) và kiểu số thực có độ chính xác gấp đôi (double).

Kiểu dữ liệu do người dùng định nghĩa bao gồm kiểu xâu kí tự (string), kiểu mảng (array), kiểu tập hợp (union), kiểu cấu trúc (struct), kiểu file, kiểu con trỏ (pointer) và các kiểu dữ liệu được định nghĩa mới hoàn toàn như kiểu danh sách móc nối (link list), kiểu cây (tree)...

Kích cỡ của kiểu cơ bản đồng nghĩa với miền xác định của kiểu với biểu diễn nhị phân của nó, và phụ thuộc vào từng hệ thống máy tính cụ thể. Để xác định kích cỡ của kiểu nên dùng toán tử sizeof(type). Chương trình sau sẽ liệt kê kích cỡ của các kiểu cơ bản.

Ví dụ 1.1. Kiểm tra kích cỡ của kiểu

```
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
#include <io.h>
void main(void) {
        printf("\n Kích cỡ kiểu kí tự: %d", sizeof(char));
        printf("\n Kích cỡ kiểu kí tự không dấu: %d", sizeof(unsigned char));
        printf("\n Kích cỡ kiểu số nguyên không dấu: %d", sizeof(unsigned int));
        printf("\n Kích cỡ kiểu số nguyên có dấu: %d", sizeof(signed int));
        printf("\n Kích cỡ kiểu số nguyên dài không dấu: %d", sizeof(unsigned long));
        printf("\n Kích cỡ kiểu số nguyên dài có dấu: %d", sizeof(signed long));
        printf("\n Kích cỡ kiểu số thực có độ chính xác đơn: %d", sizeof(float));
        printf("\n Kích cỡ kiểu số thực có độ chính xác kép: %d", sizeof(double));
        getch();
}
```

Kích cỡ của các kiểu dữ liệu do người dùng định nghĩa là tổng kích cỡ của mỗi kiểu thành viên trong nó. Chúng ta cũng vẫn dùng toán tử sizeof(tên kiểu) để xác định độ lớn tính theo byte của các kiểu dữ liệu này.

Một điểm đặc biệt chú ý trong khi lập trình trên các cấu trúc dữ liệu là cấu trúc dữ liệu nào thì phải kèm theo phép toán đó, vì một biến được gọi là thuộc kiểu dữ liệu nào đó nếu như nó nhận một giá trị từ miền xác định của kiểu và các phép toán trên kiểu dữ liệu đó.

1.3. NGUYÊN LÝ TỐI THIỂU

Hãy bắt đầu từ một tập nguyên tắc và tối thiểu các phương tiện là các cấu trúc lệnh, kiểu dữ liệu cùng các phép toán trên nó và thực hiện viết chương trình. Sau khi nắm chắc những công cụ vòng đầu mới đặt vấn đề mở rộng sang hệ thống thư viện tiện ích của ngôn ngữ.

Khi làm quen với một ngôn ngữ lập trình nào đó, không nhất thiết phải lệ thuộc quá nhiều vào hệ thống thư viện hàm của ngôn ngữ, mà điều quan trọng hơn là trước một bài toán cụ thể, chúng ta sử dụng ngôn ngữ để giải quyết nó thế nào, và phương án tốt nhất là lập trình bằng chính hệ thống thư viện hàm của riêng mình. Do vậy, đối với các ngôn ngữ lập trình, chúng ta chỉ cần nắm vững một số các công cụ tối thiểu như sau:

1.3.1. Tập các phép toán

Tập các phép toán số học: + (cộng); - (trừ); * (nhân); % (lấy phần dư); / (chia).

Tập các phép toán số học mở rộng:

++a ⇔ a = a + 1; // tăng giá trị biến nguyên a lên một đơn vị;

--a ⇔ a = a - 1; // giảm giá trị biến nguyên a xuống một đơn vị;

a+ = n ⇔ a = a+n; // tăng giá trị biến nguyên a lên n đơn vị;

a- = n ⇔ a = a - n; // giảm giá t rị biến nguyên a xuống n đơn vị;

a% = n ⇔ a = a%n; // lấy giá trị biến a modul với n;

a/ = n ⇔a = a/n; // lấy giá trị biến a chia cho n;

a* = n ⇔ a = a*n; // lấy giá trị biến a nhân với n;

Tập các phép toán so sánh: > (lớn hơn), < (nhỏ hơn), > = (lớn hơn hoặc bằng), <= (nhỏ hơn hoặc bằng), == (đúng bằng), != (khác). Qui tắc viết được thể hiện như sau:

```
if(a > b) { . . }        // nếu a lớn hơn b
if(a < b) { . . }        // nếu a nhỏ hơn b
if(a > = b) { . . }      // nếu a lớn hơn hoặc bằng b
if(a <= b) { . . }       // nếu a nhỏ hơn hoặc bằng b
if(a == b) { . . }       // nếu a đúng bằng b
if(a!= b) { . . }        // nếu a khác b
```

Tập các phép toán logic: && (và), || (hoặc), ! (phủ định)

&&: Phép và logic chỉ cho giá trị đúng khi hai biểu thức tham gia đều có giá trị đúng (giá trị đúng của một biểu thức trong ngôn ngữ lập trình C được hiểu là biểu thức có giá trị khác 0).

||: Phép hoặc logic chỉ cho giá trị sai khi cả hai biểu thức tham gia đều có giá trị sai.

!: Phép phủ định cho giá trị đúng nếu biểu thức có giá trị sai và ngược lại cho giá trị sai khi biểu thức có giá trị đúng. Ngữ nghĩa của các phép toán được minh họa thông qua các câu lệnh sau:

```
int a = 3, b = 5;
if((a != 0) && (b != 0))     // nếu a khác 0 và b khác 0
if((a != 0) || (b != 0))     // nếu a khác 0 hoặc b khác 0
if(!a)                       // phủ định a khác 0
if(a == b)                   // nếu a đúng bằng b
```

Các toán tử thao tác bit (không sử dụng cho float và double):

&: Phép hội các bit.

|: Phép tuyển các bit.

^: Phép tuyển các bit có loại trừ.

<< : Phép dịch trái (dịch sang trái n bit giá trị 0)

> > : Phép dịch phải (dịch sang phải n bit có giá trị 0)

~: Phép lấy phần bù.

Ví dụ 1.2. Viết chương trình kiểm tra các toán tử thao tác bit

```
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
#include <io.h>
void main(void) {
        unsigned int a = 3, b = 5, c; clrscr();
        c = a & b; printf("\n c = a & b = %d", c);
        c = a | b; printf("\n c = a | b = %d", c);
        c = a ^ b; printf("\n c = a ^ b = %d", c);
        c = ~a; printf("\n c = ~a = %d", c);
        c = a << b; printf("\n c = a << b = %d", c);
        c = a > > b; printf("\n c = a > > b = %d", c);
        getch();
}
```

Toán tử chuyển đổi kiểu: Ta có thể dùng toán tử chuyển đổi kiểu để nhận được kết quả tính toán như mong muốn. Qui tắc chuyển đổi kiểu được thực hiện theo qui tắc: (kiểu) biến.

Ví dụ 1.3. Tính giá trị phép chia hai số nguyên a và b

```
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
#include <io.h>
```

```
void main(void)
        int a = 3, b = 5; float c;
        c = (float) a / (float) b;
        printf("\n thương c = a / b = %6.2f", c);
        getch();
}
```

Thứ tự ưu tiên các phép toán: Khi viết một biểu thức, chúng ta cần lưu ý tới thứ tự ưu tiên tính toán các phép toán, các bảng tổng hợp sau đây phản ánh trật tự ưu tiên tính toán của các phép toán số học và phép toán so sánh.

Bảng dưới đây tổng hợp thứ tự ưu tiên tính toán các phép toán số học và so sánh.

Tên toán tử	Chiều tính toán
(), [], - >	L - > R
-, ++, --, !, ~, sizeof()	R - > L
*, /, %	L - > R
+, -	L - > R
> > , <<	L - > R
< , <= , > , > = ,	L - > R
==, !=	L - > R
&	L - > R
^	L - > R
|	L - > R
&&	L - > R
||	L - > R
?:	R - > L
= , + = , - = , * = , / = , % = , & = , ^ = , | = , <<= , > > =	R - > L
Ghi chú: R (Right): phải; L (Left): trái.	

1.3.2. Tập các lệnh vào ra cơ bản

Nhập dữ liệu từ bàn phím: scanf("format_string, ...", &parameter ...);

Nhập dữ liệu từ tệp (file): fscanf(file_pointer, "format_string, ...", &parameter, ...);

Nhận một ký tự từ bàn phím: getch(); getchar();

Nhận một ký tự từ tệp: fgetc(file_pointer, character_name);

Nhập một string (xâu) từ bàn phím: gets(string_name);

Nhận một string từ tệp text (văn bản): fgets(string_name, number_character, file_pointer);

Xuất dữ liệu ra màn hình: printf("format_string ...", parameter ...);

Xuất dữ liệu ra tệp: fprintf(file_pointer, "format_string ...", parameter ...);

Xuất một ký tự ra màn hình: putch(character_name);

Xuất một ký tự ra tệp: fputc(file_pointer, character_name);

Xuất một string ra màn hình: puts(const_string_name);

Xuất một string ra tệp: fputs(file_pointer, const_string_name);

1.3.3. Thao tác trên các kiểu dữ liệu có cấu trúc

1. Tập thao tác trên string

+ Cách tổ chức string và các thao tác trên string:

char *strchr(const char *s, int c): tìm ký tự c đầu tiên xuất hiện trong xâu s;

char *stpcpy(char *dest, const char *src): copy xâu src vào dest;

int strcmp(const char *s1, const char *s2): so sánh hai xâu s1 và s2 theo thứ tự từ điển, nếu s1 < s2 thì hàm trả lại giá trị nhỏ hơn 0. Nếu s1 > s2 hàm trả lại giá trị dương. Nếu s1 == s2 hàm trả lại giá trị 0.

char *strcat(char *dest, const char *src): thêm xâu src vào sau xâu dest.

char *strlwr(char *s): chuyển xâu s từ ký tự in hoa thành ký tự in thường.

char *strupr(char *s): chuyển xâu s từ ký tự thường hoa thành ký tự in hoa.

char *strrev(char *s): đảo ngược xâu s.

char *strstr(const char *s1, const char *s2): tìm vị trí đầu tiên của xâu s2 trong xâu s1.

int strlen(char *s): cho độ dài của xâu ký tự s.

2. *Tập thao tác trên con trỏ*

Thao tác lấy địa chỉ của biến: &parameter_name;

Thao tác lấy nội dung biến (biến có kiểu cơ bản): *pointer_name;

Thao tác trỏ tới phần tử tiếp theo: ++pointer_name;

Thao tác trỏ tới phần tử thứ n kể từ vị trí hiện tại:

```
pointer_name = pointer_name +n;
```

Thao tác trỏ tới phần tử sau con trỏ kể từ vị trí hiện tại:

```
--pointer_name;
```

Thao tác trỏ tới phần tử sau n phần tử kể từ vị trí hiện tại:

```
pointer_name = pointer_name - n;
```

Thao tác cấp phát bộ nhớ cho con trỏ:

```
void *malloc(size_t size); void *calloc(size_t nitems, size_t size);
```

Thao tác cấp phát lại bộ nhớ cho con trỏ:

```
void *realloc(void *block, size_t size);
```

Thao tác giải phóng bộ nhớ cho con trỏ: *void free(void *block);*

3. *Tập thao tác trên cấu trúc*

Định nghĩa cấu trúc:

```
struct struct_name {
        type_1 parameter_name_1;
        type_2 parameter_name_2;
        ... ... ... ... ... ... ...
        type_k parameter_name_k;
} struct_parameter_name;
```

Phép truy nhập tới thành phần cấu trúc:

```
struct_parameter_name.parameter_name.
```

Phép gán hai cấu trúc cùng kiểu:

```
struct_parameter_name_1 = struct_parameter_name_2;
```

Phép tham chiếu tới thành phần của con trỏ cấu trúc:

```
pointer_struct_parameter_name - > struct_parameter_name.
```

4. Tập thao tác trên tệp

Khai báo con trỏ tệp: `FILE * file_pointer;`

Thao tác mở tệp theo mode: `FILE *fopen(const char *filename, const char *mode);`

Thao tác đóng tệp: `int fclose(FILE *stream);`

Thao tác đọc từng dòng trong tệp: `char *fgets(char *s, int n, FILE *stream);`

Thao tác đọc từng khối trong tệp:

```
size_t fread(void *ptr, size_t size, size_t n, FILE *stream);
```

Thao tác ghi từng dòng vào tệp: `int fputs(const char *s, FILE *stream);`

Thao tác ghi từng khối vào tệp:

```
size_t fwrite(const void *ptr, size_t size, size_t n, FILE *stream);
```

Thao tác kiểm tra sự tồn tại của tệp: `int access(const char *filename, int amode);`

Thao tác đổi tên tệp: `int rename(const char *oldname, const char *newname);`

Thao tác loại bỏ tệp: `int unlink(const char *filename);`

1.4. NGUYÊN LÝ ĐỊA PHƯƠNG

- *Các biến địa phương trong hàm, thủ tục hoặc chu trình cho dù có trùng tên với biến toàn cục thì khi xử lý biến đó trong hàm hoặc thủ tục vẫn không làm thay đổi giá trị của biến toàn cục.*
- *Tên của các biến trong đối của hàm hoặc thủ tục đều là hình thức.*
- *Mọi biến hình thức truyền theo trị cho hàm hoặc thủ tục đều là các biến địa phương.*
- *Các biến khai báo bên trong các chương trình con, hàm hoặc thủ tục đều là biến địa phương.*
- *Khi phải sử dụng biến phụ nên dùng biến địa phương và hạn chế tối đa việc sử dụng biến toàn cục để tránh xảy ra các hiệu ứng phụ.*

Ví dụ hoán đổi giá trị của hai số a và b sau đây sẽ minh họa rõ hơn về nguyên lý địa phương.

Ví dụ 1.4. Hoán đổi giá trị của hai biến a và b.

```
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
#include <io.h>
int a, b;                          // khai báo a, b là hai biến toàn cục.
void Swap(void) {
        int a, b, temp;            // khai báo a, b là hai biến địa phương
        a = 3; b = 5;              // gán giá trị cho a và b
        temp = a; a = b; b = temp; // đổi giá trị của a và b
        printf("\n Kết quả thực hiện trong thủ tục a = %5d b = %5d", a, b);
```

```
}
void main(void) {
        a = 1; b = 8;                // khởi đầu giá trị cho biến toàn cục a, b.
        Swap();
        printf("\n Kết quả sau khi thực hiện thủ tục a = %5d b = %5d", a, b);
        getch();
}
```

Kết quả thực hiện chương trình:

```
Kết quả thực hiện trong thủ tục a = 5 b = 3
Kết quả sau khi thực hiện thủ tục a = 1 b = 8
```

Trong ví dụ trên a, b là hai biến toàn cục, hai biến a, b trong thủ tục Swap là hai biến cục bộ. Các thao tác trong thủ tục Swap gán cho a giá trị 3 và b giá trị 5 sau đó thực hiện đổi giá trị của a = 5 và b = 3 là công việc xử lý nội bộ của thủ tục mà không làm thay đổi giá trị của biến toàn cục của a, b sau thi thực hiện xong thủ tục Swap. Do vậy, kết quả sau khi thực hiện Swap a = 1, b = 8; điều đó chứng tỏ trong thủ tục Swap chưa bao giờ sử dụng tới hai biến toàn cục a và b. Tuy nhiên, trong ví dụ sau, thủ tục Swap lại làm thay đổi giá trị của biến toàn cục a và b vì nó thao tác trực tiếp trên biến toàn cục.

Ví dụ 1.5. Đổi giá trị của hai biến a và b

```
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
#include <io.h>
int a, b;                        // khai báo a, b là hai biến toàn cuc.
void Swap(void) {
        int temp;                // khai báo a, b là hai biến địa phương
        a = 3; b = 5;            // gán giá trị cho a và b
```

```
        temp = a; a = b; b = temp;          // đổi giá trị của a và b
        printf("\n Kết quả thực hiện trong thủ tục a = %5d b = %5d", a, b);
}
void main(void) {
        a = 1; b = 8;                       // khởi đầu giá trị cho biến toàn cục a, b.
        Swap();
        printf("\n Kết quả sau khi thực hiện thủ tục a = %5d b = %5d", a, b);
        getch();
}
```

Kết quả thực hiện chương trình:

```
Kết quả thực hiện trong thủ tục a = 8 b = 1
Kết quả sau khi thực hiện thủ tục a = 1 b = 8
```

1.5. NGUYÊN LÝ NHẤT QUÁN

- *Dữ liệu thế nào thì phải thao tác thế ấy. Cần sớm phát hiện những mâu thuẫn giữa cấu trúc dữ liệu và thao tác để kịp thời khắc phục.*

Như chúng ta đã biết, kiểu là một tên chỉ tập các đối tượng thuộc miền xác định cùng với những thao tác trên nó. Một biến khi định nghĩa bao giờ cũng thuộc một kiểu xác định nào đó hoặc là kiểu cơ bản hoặc kiểu do người dùng định nghĩa. Thao tác với biến phụ thuộc vào những thao tác được phép của kiểu. Hai kiểu khác nhau được phân biệt bởi tên, miền xác định và các phép toán trên kiểu dữ liệu. Tuy nhiên, trên thực tế có nhiều lỗi nhập nhằng giữa phép toán và cấu trúc dữ liệu mà chúng ta cần hiểu rõ.

Đối với kiểu ký tự, về nguyên tắc chúng ta không được phép thực hiện các phép toán số học trên nó, nhưng ngôn ngữ C luôn đồng nhất giữa ký tự với số nguyên có độ lớn 1 byte. Do vậy, những phép toán số học trên các ký tự thực chất là những phép toán số học trên các số

nguyên. Chẳng hạn, những thao tác như trong khai báo dưới đây là được phép:

```
char x1 = 'A', x2 = 'z';
x1 = (x1 + 100) % 255;
x2 = (x2-x1) %255;
```

Mặc dù x1, x2 được khai báo là hai biến kiểu char, nhưng trong thao tác:

```
x1 = (x1 + 100) % 255;
x2 = (x2 +x1) % 255;
```

Chương trình dịch sẽ tự động chuyển đổi x1 thành mã của ký tự 'A' là 65, x2 thành mã ký tự 'z' là 122 để thực hiện phép toán. Kết quả nhận được x1 là một ký tự có mã là (65+100)%255 = 165; x2 là ký tự có mã là 32 ứng với mã của ký tự trống (space).

Chúng ta có thể thực hiện được các phép toán số học trên kiểu *int*, *long*, *float*, *double*. Nhưng đối với *int* và *long*, chúng ta cần đặc biệt chú ý phép chia hai số nguyên cho ta một số nguyên, tích hai số nguyên cho ta một số nguyên, tổng hai số nguyên cho ta một số nguyên mặc dù thương hai số nguyên là một số thực, tích hai số nguyên hoặc tổng hai số nguyên có thể là một số *long int*. Do vậy, muốn nhận được kết quả đúng, chúng ta cần phải chuyển đổi các biến thuộc cùng một kiểu trước khi thực hiện phép toán. Ngược lại, ta không thể lấy modul của hai số thực hoặc thực hiện các thao tác dịch chuyển bit trên nó, vì những thao tác đó không nằm trong định nghĩa của kiểu.

Điều tương tự cũng xảy ra với các string. Trong Pascal, phép toán so sánh hai string hoặc gán trực tiếp hai record cùng kiểu với nhau là được phép, ví dụ: str1 > str2, str1:= str2; Nhưng trong C thì các phép toán trên lại không được định nghĩa, nếu muốn thực hiện nó, chúng ta chỉ có cách định nghĩa lại hoặc thực hiện nó thông qua các lời gọi

hàm. Ví dụ đơn giản sau sẽ minh họa cho những lỗi thường xảy ra sự nhập nhằng giữa phép toán và cấu trúc dữ liệu.

Ví dụ 1.6. Viết chương trình tính tổng, hiệu, tích, thương của hai số nguyên a và b.

```
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
long int tong(int a, int b) {
        return(a+b);
}
int hieu(int a, int b) {
        return(a-b);
}
long int tich(int a, int b) {
        return(a*b);
}
float thuong(int a, int b) {
        return(a/b);
}
void main(void) {
        int a = 30000, b = 20000; // khai báo và gán giá trị hai số nguyên a, b
        printf("\n Tổng hai số nguyên a + b = %ld", tong(a, b));
        printf("\n Hiệu hai số nguyên a - b = %d", hieu(a, b));
        printf("\n Tích hai số nguyên a * b = %ld", tich(a, b));
        printf("\n Thương hai số nguyên a / b = %f", thuong(a, b));
        getch();
}
```

Chương trình đơn giản được thể hiện hết sức hoàn hảo, nhưng người lập trình không ngờ rằng kết quả lại hiển thị sai như dưới đây:

```
Tổng hai số nguyên a + b = 1229648
Hiệu hai số nguyên a - b = 10000
Tích hai số nguyên a * b = 1197568
Thương hai số nguyên a / b = 1.000000
```

Trong ví dụ trên, hàm tong(30000, 20000) cho ta một số unsigned int là giá trị là tổng của hai số nguyên dương, vì tổng của hai số nguyên dương có thể vượt quá kích cỡ kiểu int để trở thành một số unsigned int, điều đó cũng xảy ra tương tự đối với hàm tich(30000, 20000); Do đó, việc thực hiện việc gán một số nguyên dương thế này sẽ cho ta kết quả không mong muốn.

Với hàm thuong(a, b) thì hoàn toàn ngược lại đối với một số ngôn ngữ (C, C++). Khi chúng ta phát biểu thương của hai số nguyên dương là một số thực được tính là a/b ($b \neq 0$), nhưng trong thực tế chương trình dịch của C lại hiểu thương của hai số nguyên cho ta kết quả là một số nguyên, do đó chúng ta nhận được kết quả không mong muốn. Giải pháp để khắc phục mâu thuẫn giữa cấu trúc dữ liệu và thao tác trên cấu trúc dữ liệu có thể thực hiện bằng cách chuyển đổi kiểu của đối truyền cho hàm như phiên bản sau.

Ví dụ 1.7. Viết chương trình tính tổng, hiệu, tích, thương của hai số nguyên a và b.

```
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
long int tong(int a, int b) {
        return((long) a + (long) b);
}
int hieu(int a, int b) {
```

```
        return(a-b);
}
long int tich(int a, int b) {
        return((long) a * (long) b);
}
float thuong(int a, int b) {
        float k;
        k = (float) a / (float) b;
        return(k);
}
void main(void) {
        int a = 30000, b = 20000;          // khai báo và gán giá trị hai số nguyên a, b
        printf("\n Tổng hai số nguyên a + b = %ld", tong(a, b));
        printf("\n Hiệu hai số nguyên a - b = %d", hieu(a, b));
        printf("\n Tích hai số nguyên a * b = %ld", tich(a, b));
        printf("\n Thương hai số nguyên a/b = %f", thuong(a, b));
        getch();
}
```

Kết quả thực hiện chương trình:

```
Tổng hai số nguyên a + b = 50000
Hiệu hai số nguyên a - b = 10000
Tích hai số nguyên a * b = 600000000
Thương hai số nguyên a/b = 1.500000
```

Tóm lại, cần nắm vững nguyên tắc, định nghĩa và những qui định riêng của ngôn ngữ cho từng kiểu dữ liệu và các phép toán trên nó để đảm bảo tính nhất quán trong khi xử lý dữ liệu.

1.6 NGUYÊN LÝ AN TOÀN

- *Lỗi nặng nhất nằm ở mức cao nhất (mức ý đồ thiết kế) và ở mức thấp nhất thủ tục phải chịu tải lớn nhất.*
- *Mọi lỗi, dù là nhỏ nhất cũng phải được phát hiện ở một bước nào đó của chương trình. Quá trình kiểm tra và phát hiện lỗi phải được thực hiện trước khi lỗi đó hoành hành.*

Các loại lỗi thường xảy ra trong khi viết chương trình có thể được tổng kết lại như sau:

Lỗi được thông báo bởi từ khoá error (lỗi cú pháp): loại lỗi này thường xảy ra trong khi soạn thảo chương trình, chúng ta có thể viết sai các từ khoá ví dụ thay vì viết là *int* chúng ta soạn thảo sai thành *Int* (lỗi chữ in thường thành in hoa), hoặc viết sai cú pháp các biểu thức như thiếu các dấu ngoặc đơn, ngoặc kép hoặc dấu chấm phảy khi kết thúc một lệnh, hoặc chưa khai báo nguyên mẫu cho hàm.

Lỗi được thông báo bởi từ khoá warning (lỗi cảnh báo): lỗi này thường xảy ra khi ta khai báo biến trong chương trình nhưng lại không sử dụng tới chúng, hoặc lỗi trong các biểu thức kiểm tra khi biến được kiểm tra không xác định được giá trị của nó, hoặc lỗi do thứ tự ưu tiên các phép toán trong biểu thức. Hai loại lỗi error và warning được thông báo ngay khi dịch chương trình thành tệp *.OBJ. Quá trình liên kết (linker) các file *.OBJ để tạo nên file chương trình mã máy *.EXE chỉ được tiếp tục khi chúng ta hiệu đính và khử bỏ mọi lỗi error.

Lỗi xảy ra trong quá trình liên kết (linker): lỗi này thường xuất hiện khi ta sử dụng tới các lời gọi hàm, nhưng những hàm đó mới chỉ tồn tại dưới dạng nguyên mẫu (function prototype) mà chưa được mô tả chi tiết các hàm, hoặc những lời hàm gọi chưa đúng với tên của nó. Lỗi này được khắc phục khi ta bổ sung đoạn chương trình con mô tả chi tiết cho hàm hoặc sửa đổi lại những lời gọi hàm tương ứng.

Ta quan niệm, lỗi cú pháp (error), lỗi cảnh báo (warning) và lỗi liên kết (linker) là lỗi tầm thường vì những lỗi này đã được trình biên dịch (Compiler) của các ngôn ngữ lập trình phát hiện được. Để khắc phục các lỗi loại này, chúng ta chỉ cần phải đọc và hiểu được những thông báo lỗi thường được viết bằng tiếng Anh. Cũng cần phải lưu ý rằng, do mức độ phức tạp của chương trình dịch nên không phải lỗi nào cũng được chỉ ra một cách tường minh và chính xác hoàn toàn tại nơi xuất hiện lỗi.

Loại lỗi cuối cùng mà các trình biên dịch không thể phát hiện nổi đó là lỗi do chính lập trình viên gây nên trong khi thiết kế chương trình và xử lý dữ liệu. Những lỗi này không được compiler thông báo mà nó phải trả giá bằng quá trình tự kiểm tra thử (test) hoặc chứng minh được tính đúng đắn của chương trình. Lỗi có thể nằm ở chính ý đồ thiết kế, hoặc lỗi do không lường trước được tính chất của mỗi loại thông tin vào. Ví dụ sau minh họa cho lỗi thường xảy ra thuộc loại này.

Ví dụ 1.8. Tính tổng hai đa thức A bậc n, đa thức B bậc m.

```
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
#define MAX 100
typedef float dathuc[MAX];
void In(dathuc A, int n, char c) {
        int i;
        printf("\n Đa thức %c: ", c);
        for(i = 0; i < n; i++)
                printf("%6.2f", A[i]);
}
void Init(dathuc A, int *n, dathuc B, int *m) {
        int i; float temp;
```

```
        printf("\n Nhập n = "); scanf("%d", n); *n = *n+1;
        printf("\n Nhập m = "); scanf("%d", m); *m = *m+1;
        printf("\n Nhập hệ số đa thức A: ");
        for(i = 0; i < *n; i++) {
                printf("\n A[%d] = ", i); scanf("%f", &temp);
                A[i] = temp;
        }
        printf("\n Nhập hệ số đa thức B: ");
        for(i = 0; i < *m; i++) {
                printf("\n B[%d] = ", i); scanf("%f", &temp);
                B[i] = temp;
        }
        In(A, *n, 'A'); In(B, *m, 'B');
}
void Tong(dathuc A, int n, dathuc B, int m, dathuc C) {
        int i, k;
        if(n > = m) {
                k = n;
                for(i = 0; i < m; i++)
                        C[i] = A[i]+B[i];
                for(i = m; i < n; i++)
                        C[i] = A[i];
                In(C, k, 'C');
        }
        else {
                k = m;
                for(i = 0; i < n; i++)
                        C[i] = A[i]+B[i];
```

```
            for(i = n; i < m; i++)
                C[i] = B[i];
            In(C, k, 'C');
        }
}
void main(void) {
        dathuc A, B, C;
        int n, m;
        Init(A, &n, B, &m);
        Tong(A, n, B, m, C);
}
```

Trong ví dụ trên, chúng ta sử dụng định nghĩa MAX = 100 để giải quyết bài toán. MAX được hiểu là bậc của đa thức lớn nhất mà chúng ta cần xử lý. Như vậy, bản thân việc định nghĩa MAX đã hạn chế tới phạm vi bài toán, hạn chế đó cũng có thể xuất phát từ ý đồ thiết kế. Do vậy, nếu người sử dụng nhập n > MAX thì chương trình sẽ gặp lỗi. Nếu chúng ta khắc phục bằng cách định nghĩa MAX đủ lớn thì trong trường hợp xử lý các đa thức có bậc n nhỏ sẽ gây nên hiện tượng lãng phí bộ nhớ, và trong nhiều trường hợp không đủ bộ nhớ để định nghĩa đa thức. Giải pháp khắc phục các lỗi loại này là chúng ta sử dụng con trỏ thay cho các hằng, kỹ thuật này sẽ được thảo luận kỹ trong Chương 2.

1.7. PHƯƠNG PHÁP TOP-DOWN (TỪ TRÊN XUỐNG)

- *Quá trình phân tích bài toán được thực hiện từ trên xuống dưới. Từ vấn đề chung nhất đến vấn đề cụ thể nhất. Từ mức trừu tượng mang tính chất tổng quan tới mức đơn giản nhất là đơn vị chương trình.*

Một trong những nguyên lý quan trọng của lập trình cấu trúc là phương pháp phân tích từ trên xuống (Top - Down) với quan điểm

"thấy cây không bằng thấy rừng", phải đứng cao hơn để quan sát tổng thể khu rừng chứ không thể đứng trong rừng quan sát chính nó.

Quá trình phân rã bài toán được thực hiện theo từng mức khác nhau. Mức thấp nhất được gọi là mức tổng quan (level 0), mức tổng quan cho phép ta nhìn tổng thể hệ thống thông qua các chức năng của nó, nói cách khác mức 0 sẽ trả lời thay cho câu hỏi "Hệ thống có thể thực hiện được những gì?". Mức tiếp theo là mức các chức năng chính. Ở mức này, những chức năng cụ thể được mô tả. Một hệ thống có thể được phân tích thành nhiều mức khác nhau, mức thấp được phép sử dụng các dịch vụ của mức cao. Quá trình phân tích tiếp tục phân rã hệ thống theo từng chức năng phụ cho tới khi nào nhận được mức các đơn thể (UNIT, Function, Procedure), khi đó chúng ta tiến hành cài đặt hệ thống.

Chúng ta sẽ làm rõ hơn từng mức của quá trình Top-Down thông qua bài toán sau:

Bài toán: Cho hai số nguyên có biểu diễn nhị phân là $a = (a_1, a_2, ..., a_n)$, $b = (b_1, b_2, .., b_n)$; $a_i, b_i = 0, 1, i = 1, 2, ... n$. Hãy xây dựng tập các thao tác trên hai số nguyên đó.

Mức tổng quan (level 0):

Hình dung toàn bộ những thao tác trên hai số nguyên $a = (a_1, a_2, ..., a_n)$, $b = (b_1, b_2, .., b_n)$ với đầy đủ những chức năng chính của nó. Giả sử những thao tác đó bao gồm:

F1 - Chuyển đổi a, b thành các số nhị phân;

F2 - Tính tổng hai số nguyên: a + b;

F3 - Tính hiệu hai số nguyên: a - b;

F4 - Tính tích hai số nguyên: a * b;

F5 - Thương hai số nguyên: a/b;

F6 - Phần dư hai số nguyên: a % b;

F7 - Ước số chung lớn nhất của hai số nguyên.

Mức 1. Mức các chức năng chính: mỗi chức năng cần mô tả đầy đủ thông tin vào (Input), thông tin ra (Output), khuôn dạng (Format) và các hành động (Actions).

Chức năng F1: Chuyển đổi a, b thành các số ở hệ nhị phân

```
Input   :     a: integer;
Output  :     a = (a1, a2, ..., an)b;     (* khai triển cơ số b bất kỳ *)
Format  :     Binary(a);
Actions
        Q = n; k = 0;
        while Q≠ 0
        begin
                ak = q mod b;
                q = q div b;
                k = k +1;
        end;
< Khai triển cơ số b của a là (ak-1, ak-2, . ., a1, a0) > ;
EndAction;
```

Chức năng F2: Tính tổng hai số nguyên a, b.

```
Input:
        a = (a1, a2, ..., an),
        b = (b1, b2, .., bn);
Output:
        c = a + b;
Format: Addition(a, b);
Actions
        c = 0;
        for j = 0 to n-1 do
```

begin

$d = (a_j + b_j + c)$ div 2;

$s_j = a_j + b_j + c - 2d$;

$c = d$;

end;

$s_n = c$;

< Khai triển nhị phân của tổng là $(s_n s_{n-1} ... s_1, s_0)_2$ >

EndAction;

Chức năng F3: Hiệu hai số nguyên a, b.

Input:

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

$b = (b_1, b_2, .., b_n)$;

Output:

$c = a - b$;

Format: Subtraction(a, b);

Actions

$b = -b$;

c = Addition(a, b);

return(c);

EndAction;

Chức năng F4: Tích hai số nguyên a, b.

Input:

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

$b = (b_1, b_2, .., b_n)$;

Output:

$c = a * b$;

Format: Multual(a, b);

Actions

```
        for j = 0 to n-1 do
        begin
                if b_j = 1 then c_j = a << j
                else c_j = 0;
        end;
        (* c_0, c_1, ..., c_n-1 là các tích riêng *)
        p := 0;
        for j = 0 to n-1 do
                p = Addition(p, c_j);
        return(p);
EndAction;
```

Chức năng F5: Thương hai số nguyên a, b.

```
Input:
        a = (a_1, a_2, ..., a_n),
        b = (b_1, b_2, .., b_n);
Output:
        c = a div b;
Format: Division(a, b);
Actions
        c = 0;
        while(a > = b) do
        begin
                c = c +1;
                a = Subtraction(a, b);
        end;
        return(c);
EndAction;
```

Chức năng F6: Modul hai số nguyên a, b.

```
Input:
        a = (a1, a2, ..., an),
        b = (b1, b2, .., bn);
Output:
        c = a mod b;
Format: Modulation(a, b);
Actions
        while (a > = b) do
                a = Subtraction(a, b);
        return(a);
EndAction;
```

Chức năng F7: Ước số chung lớn nhất hai số nguyên a, b.

```
Input:
        a = (a1, a2, ..., an),
        b = (b1, b2, .., bn);
Output:
        c = USCLN(a, b);
Format: USCLN(a, b);
Actions
        while (a≠ b) do
        begin
                if a > b then
                        a = Subtraction(a, b)
                else
                        b = Subtraction(b, a);
        return(a);
EndAction;
```

Để ý rằng, sau khi phân rã bài toán ở mức 1, chúng ta chỉ cần xây dựng hai phép toán cơ bản cho các số nguyên a và b, đó là phép tính cộng và phép tính nhân các số nhị phân của a và b. Vì hiệu hai số a và b chính là tổng số của (a, -b). Tương tự như vậy, tích hai số a và b được biểu diễn bằng tổng của một số lần phép nhân một bit nhị phân của với a. Phép chia và lấy phần dư hai số a và b chính là phép trừ nhiều lần số a. Phép tìm USCLN cũng tương tự như vậy.

Đối với các hệ thống lớn, quá trình còn được mô tả tiếp tục cho tới khi nhận được mức đơn vị chương trình. Trong ví dụ đơn giản này, mức đơn vị chương trình xuất hiện ngay tại mức 1 nên chúng ta không cần phân rã tiếp nữa mà dừng lại để cài đặt hệ thống.

1.8. PHƯƠNG PHÁP BOTTOM - UP

- *Đi từ cái riêng tới cái chung, từ các đối tượng thành phần ở mức cao tới các đối tượng thành phần ở mức thấp, từ mức đơn vị chương trình tới mức tổng thể, từ những đơn vị đã biết lắp đặt thành những đơn vị mới.*

Nếu như phương pháp Top - Down là phương pháp phân rã vấn đề một cách có hệ thống từ trên xuống, được ứng dụng chủ yếu cho quá trình phân tích và thiết hệ thống, thì phương pháp Bottom - Up thường được sử dụng cho quá trình cài đặt hệ thống. Trong ví dụ trên, chúng ta sẽ không thể xây dựng được chương trình một cách hoàn chỉnh nếu như ta chưa xây dựng được các hàm Binary(a), Addition(a, b), Subtraction(a, b), Multial(a, b), Division(a, b), Modulation(a, b), USCLN(a, b). Chương trình sau thể hiện quá trình cài đặt chương trình theo nguyên lý Bottom - Up:

```
#include <stdio.h>

#include <math.h>

#include <conio.h>

#include <stdlib.h>
```

```
#include <alloc.h>
#include <dos.h>
void Init(int *a, int *b) {
        printf("\n Nhập a = "); scanf("%d", a);
        printf("\n Nhập b = "); scanf("%d", b);
}
void Binary(int a) {
        int i, k = 1;
        for(i = 15; i > = 0; i--) {
                if(a & (k << i))
                        printf("%2d", 1);
                else
                        printf("%2d", 0);
        }
        printf("\n"); delay(500);
}
int bit(int a, int k) {
        int j = 1;
        if(a & (j << k))
                return(1);
        return(0);
}
int Addition(int a, int b) {
        int du, d, s, j, c = 0;
        du = 0;
        for(j = 0; j <= 15; j++) {
                d = (bit(a, j) + bit(b, j) +du)/2;
                s = bit(a, j)+bit(b, j)+ du - 2*d;
```

```
                c = c | (s << j);
                du = d;
        }
        return(c);
}
int Multial(int a, int b) {
        int c, j, p = 0;
        for(j = 0; j <= 15; j++) {
                c = bit(b, j);
                if(c == 1) {
                        c = a << j;
                        p = Addition(p, c);
                }
                else c = 0;
        }
        return(p);
}
int Subtraction(int a, int b) {
        int c;
        b = -b;
        c = Addition(a, b); return(c);
}
int Modul(int a, int b) {
        while(a > = b)
                a = Subtraction(a, b);
        return(a);
}
int Division(int a, int b) {
```

```
        int d = 0;
        while(a > = b) {
                a = Subtraction(a, b);
                d++;
        }
        return(d);
}
int USCLN(int a, int b) {
        while(a != b) {
                if(a > b) a = Subtraction(a, b);
                else b = Subtraction(b, a);
        }
        return(a);
}
void main(void) {
        int a, b, key, control = 0;

        do {
                clrscr();
                printf("\n Tập thao tác với số nguyên");
                printf("\n 1 - Nhập hai số a, b");
                printf("\n 2 - Số nhị phân của a, b");
                printf("\n 3 - Tổng hai số a, b");
                printf("\n 4 - Hiệu hai số a, b");
                printf("\n 5 - Tích hai số a, b");
                printf("\n 6 - Thương hai số a, b");
                printf("\n 7 - Phần dư hai số a, b");
                printf("\n 8 - USCLN hai số a, b");
```

```
printf("\n 0 - Trở về");
key = getch();
switch(key) {
        case '1': Init(&a, &b); control = 1; break;
        case '2':
                if(control) {
                        Binary(a); Binary(b);
                }
                break;
        case '3':
                if(control)
                    printf("\n Tổng a + b = %d", Addition(a, b));
                break;
        case '4':
                if(control)
                    printf("\n Hiệu a - b = %d", Subtraction(a, b));
                break;
        case '5':
                if(control)
                    printf("\n Tích a * b = %d", Multial(a, b));
                break;
        case '6':
                if(control)
               printf("\n Chia nguyên a div b = %d", Division(a, b));
                break;
        case '7':
                if(control)
                    printf("\n Phần dư a mod b = %d", Modul(a, b));
```

```
                    break;
            case '8':
                    if(control)
                printf("\n Ước số chung lớn nhất: %d", USCLN(a, b));
                    break;
        }
        delay(1000);
    } while(key != '0');
}
```

BÀI TẬP CHƯƠNG 1

1.1. Viết chương trình giải phương trình bậc 2.

1.2. Tìm các nghiệm nguyên dương của hệ phương trình:

$X + Y + Z = 100$

$5X + 3Y + Z/3 = 100$

1.3. Cho số tự nhiên n. Hãy tìm tất cả các bộ 3 các số tự nhiên a, b, c sao cho $a^2 + b^2 = c^2$ trong đó $a \le b \le c \le n$.

1.4. Cho số tự nhiên n. Hãy tìm các số Fibonaci nhỏ hơn n. Trong đó các số Fibonaci được định nghĩa như sau:

$U_0 = 0$; $U_1 = 1$; $U_k = U_{k-1} + U_{k-2}$; $k = 1, 2, ...$

1.5. Chứng minh rằng, với mọi số nguyên dương N, $0 < N \le 39$ thì $N^2 + N + 41$ là một số nguyên tố. Điều khẳng định trên không còn đúng với $N > 39$.

1.6. Cho số tự nhiên n. Hãy liệt kê tất cả các số nguyên tố nhỏ hơn n.

1.7. Cho số tự nhiên n. Hãy tìm tất cả các số nguyên tố nhỏ hơn n bằng phương pháp sàng Estheven.

1.8. Cho số tự nhiên n. Dùng phương pháp sàng Estheven để tìm 4 số nguyên tố bé hơn n nằm trong cùng bậc chục (ví dụ: 11, 13, 15, 17).

1.9. Cho số tự nhiên n. Hãy liệt kê tất cả các cặp số p, 4p + 1 đều là số nguyên tố nhỏ hơn n. Trong đó p cũng là số nguyên tố nhỏ hơn n.

1.10. Hãy liệt kê tất cả các số nguyên tố có 5 chữ số sao cho tổng số các chữ số trong số nguyên tố đó đúng bằng S cho trước $1 \le S \le 45$.

1.11. Một số được gọi là số Mersen nếu nó là số nguyên tố được biểu diễn dưới dạng 2^P - 1 trong đó P cũng là một số nguyên tố. Cho số tự nhiên n, tìm tất cả các số Mersen nhỏ hơn n.

1.12. Cho số tự nhiên n. Hãy phân tích n thành tích các thừa số nguyên tố. Ví dụ 12 = 2*2*3.

1.13. Hai số tự nhiên a, b được gọi là "hữu nghị" nếu tổng các ước số thực sự của a (kể cả 1) bằng b và ngược lại. Cho hai số tự nhiên P, Q. Hãy tìm tất cả các cặp số hữu nghị trong khoảng [P, Q].

1.14. Cho số tự nhiên n. Hãy tìm tất cả các số 1, 2, ..., n sao cho các số trùng với phần cuối bình phương chính nó (Ví dụ: $6^2 = 36$, $25^2 = 625$).

1.15. Một số tự nhiên được gọi là số Amstrong nếu tổng các lũy thừa bậc n của các chữ số của nó bằng chính số đó. Trong đó n là số các chữ số (Ví dụ $153 = 1^3 + 2^3 + 3^3$). Hãy tìm tất cả các số Amstrong gồm 2, 3, 4 chữ số.

1.16. Một số tự nhiên là Palindrom nếu các chữ số của nó viết theo thứ tự ngược lại thì số tạo thành là chính số đó (Ví dụ: 4884, 393). Hãy tìm:

- Tất cả các số tự nhiên nhỏ hơn 100 mà khi bình phương lên thì cho một Palindrom.
- Tất cả các số Palindrom bé hơn 100 mà khi bình phương lên chúng cho một Palindrom.

1.17. Để ghi nhãn cho những chiếc ghế trong một giảng đường, người ta sử dụng 4 ký tự, ký tự đầu tiên là một chữ cái in hoa, ba ký tự tiếp theo là một số nguyên dương không vượt quá 100. Hỏi bằng cách đó có nhiều nhất bao nhiêu chiếc ghế được đánh nhãn và đó là những nhãn nào.

1.18. Dự án đánh số điện thoại của Bang Florida được qui định như sau. Trong dự án đánh số điện thoại gồm 10 chữ số được chia

thành nhóm: mã vùng gồm 3 chữ số, nhóm mã chi nhánh gồm 3 chữ số và nhóm mã máy gồm 4 chữ số. Vì những nguyên nhân kỹ thuật nên có một số hạn chế đối với các chữ số, giả sử X biểu thị các chữ số nhận giá trị từ 0 đến 9, Y là các chữ số nhận giá trị hoặc 0 hoặc là 1, N là các chữ số nhận giá trị từ 2 đến 9. Hỏi với cách đánh số dạng NYX NNX XXXX và NXX NXX XXXX sẽ có bao nhiêu số điện thoại khác nhau. In ra màn hình và ghi lại vào tệp DT.TXT số điện thoại của vùng có mã 200, mã chi nhánh 250 và số điện thoại bắt đầu là số 9. Mỗi số điện thoại được ghi trên một dòng, mỗi dòng được ghi làm 3 phần (Mã vùng, mã chi nhánh, số điện thoại) mỗi phần phân biệt với nhau bởi một hoặc và dấu trống.

1.19. Số các máy chủ của một trung tâm máy tính được biểu diễn như một xâu bit nhị phân có độ dài k (không kể xâu rỗng). Số các máy tính có thể quản lý được của một máy chủ trong Trung tâm máy tính được hiểu là một xâu bit nhị phân có độ dài n, mỗi xâu bit nhị phân có độ dài k + n bit nhị phân được hiểu là địa chỉ truy nhập tới một máy chủ nào đó của Trung tâm máy tính. Trong đó, k bit đầu là địa chỉ máy chủ, n bit cuối là địa chỉ của máy trạm. Hỏi mạng có thể liên kết được tối đa là bao nhiêu máy tính nếu Trung tâm dùng k máy chủ và mỗi máy chủ có thể liên kết được n máy tính thành phần, in ra màn hình và ghi lại kết quả địa chỉ của các máy tính thành phần mà mỗi máy chủ quản lý vào tệp KETQUA.TXT trong đó: dòng thứ nhất ghi lại hai số k và n mỗi số được phân cách bởi một hoặc vài dấu trống, các dòng tiếp theo ghi lại địa chỉ của từng máy tính. Ví dụ với k = 2, n = 4, tệp KETQUA.TXT có dạng như sau:

```
2       4
01      0000        11  0000
01      0001        11  0001
01      0010        11  0010
```

01	0011	11	0011
01	0100	11	0100
01	0101	11	0101
01	0110	11	0101
01	0111	11	0111
01	1000	11	1000
01	1001	11	1001
01	1010	11	1010
01	1011	11	1011
01	1100	11	1100
01	1101	11	1101
01	1110	11	1110
01	1111	11	1111

1.20. Cho xâu kí tự có dạng mm/dd/yy (tháng, ngày, năm). Hãy biểu diễn biểu thức ngày tháng năm thành số nguyên 2 byte trong đó 7 bit đầu tiên biểu diễn giá trị hai số cuối của năm (00-99), 5 bit tiếp theo biểu diễn giá trị của ngày (00-31), 4 bit cuối cùng biểu diễn giá trị của tháng. Ví dụ biểu thức "12/31/99" được chuyển thành số nguyên có biểu diễn nhị phân "1100.11111.1100011".

1.21. Cho tệp dữ liệu TEXT.TXT được tổ chức thành từng dòng, độ dài tối đa của mỗi dòng là 80 kí tự. Kỹ thuật mã hoá tuyến tính là phương pháp biến đổi mã của các kí tự từ [a .. z], [A .. Z] thành một kí tự mới mà mã của nó cộng thêm với một hằng số k cho trước. Quá trình giải mã được thực hiện ngược lại. Hãy viết chương trình mô tả phương pháp mã hoá và giải mã tuyến tính tệp dữ liệu TEXT.TXT. Quá trình mã hoá được ghi lại trong tệp MAHOA.TXT, quá trình giải mã ghi lại trong tệp GIAIMA.TXT.

1.22. Cho tệp dữ liệu TEXT.TXT được tổ chức thành từng dòng, độ dài tối đa của mỗi dòng là 80 kí tự. Kỹ thuật mã hoá chẵn lẻ là phương pháp biến đổi mã của các kí tự [a . . z], [A . . Z]. Trong đó, nếu kí tự có số các bit 1 là lẻ ta bổ sung thêm một bit có giá trị một vào bit số 7 của kí tự để kí tự trở thành chẵn. Quá trình giải mã được thực hiện ngược lại. Hãy viết chương trình mô tả kỹ thuật mã hoá chẵn lẻ tệp dữ liệu TEXT.TXT. Quá trình mã hoá được ghi lại trong tệp MAHOA.TXT, quá trình giải mã ghi lại trong tệp GIAIMA.TXT.

1.23. Cho tệp dữ liệu TEXT.TXT được tổ chức thành từng dòng, độ dài tối đa của mỗi dòng là 80 kí tự. Kỹ thuật mã hoá Caresa là phương pháp biến đổi mã của các kí tự [A .. Z] thành một số ở hệ cơ số 26, lấy số nhận được cộng thêm với một hằng số k cho trước và lấy modul với 26, lấy số nhận được dịch ngược lại thành kí tự để có được một bản mã hoá chính thức. Quá trình giải mã được thực hiện ngược lại. Hãy viết chương trình mô tả kỹ thuật mã hoá Caresa tệp dữ liệu TEXT.TXT. Quá trình mã hoá được ghi lại trong tệp MAHOA.TXT, quá trình giải mã ghi lại trong tệp GIAIMA.TXT.

1.24. Cho hai số nguyên dương a, b bất kỳ được nhập từ bàn phím. Hãy viết chương trình thực hiện những thao tác sau trên hai số nguyên:

- Chuyển đổi a, b thành các số ở hệ cơ số p bất kỳ ($1 \le p \le 16$);
- Tìm tổng hai số nguyên a, b theo thuật toán cộng;
- Tìm tích hai số nguyên theo thuật toán nhân;
- Tìm ước số chung lớn nhất của a và b theo thuật toán Ơclít;

1.25. Cho byte kí tự C bất kỳ được nhập từ bàn phím $C \in$ ['A' .. 'Z'] $\cup$ ['a' .. 'z']. Hãy viết chương trình thực hiện những thao tác sau trên byte ký tự:

- In ra mã của kí tự C;
- Một byte ký tự được gọi là byte chẵn nếu tổng số các bit có giá trị 1 trong nó là chẵn, một byte được gọi là lẻ nếu tổng số các bit 1 trong nó là lẻ. Hãy kiểm tra tính chẵn lẻ của byte ký tự C;
- Kỹ thuật mã hoá chẵn lẻ cho các ký tự được thực hiện trên các byte ký tự lẻ bằng cách bổ sung giá trị 1 vào bit số 7 của ký tự để cho byte ký tự trở thành chẵn. Hãy mã hoá ký tự C theo kỹ thuật chẵn lẻ (parity);
- Kỹ thuật giải mã parity được thực hiện trên các byte chẵn mà bit số 7 của nó có giá trị 1. Hãy thực hiện giải mã ký tự C đã được mã hoá trước đó.

1.26. Cho hai xâu kí tự str1, str2 bất kỳ được nhập từ bàn phím. Hãy thực hiện những thao tác sau trên hai xâu ký tự:

- Tìm cách loại bỏ các ký tự trống (space), những ký tự xuất hiện hơn một lần trong mỗi xâu để nhận được hai xâu ký tự mới không chứa ký tự trống (space), không có hai ký tự giống nhau trong mỗi xâu.
- Tìm giao của hai xâu str1, str2 theo định nghĩa tập hợp.
- Tìm hiệu của hai xâu str1, str2 theo định nghĩa tập hợp.
- Tìm hợp của hai xâu str1, str2 theo định nghĩa tập hợp.
- Sắp xếp hợp của hai xâu theo tứ tự tăng dần.
- Tìm kí tự X bất kì có thuộc hợp của hai xâu hay không, nếu X thuộc hợp của hai xâu thì đưa ra vị trí của nó trong hợp của hai xâu.

1.27. Cho tệp dữ liệu INPUT.TXT được tổ chức như sau: dòng thứ nhất ghi lại hai số nguyên dương n là số các phần tử của dãy thực A[n]; dòng tiếp theo ghi lại n số thực là dãy các số thực A.

Hãy viết chương trình in ra màn hình và ghi lại kết quả vào tệp OUTPUT.TXT các kết quả sau:

- Dòng đầu tiên là số lớn thứ nhất và chỉ số của số lớn thứ nhất, số lớn thứ nhì và chỉ số của số lớn thứ nhì trong dãy các số A[n].
- Dòng kế tiếp là tổng các số dương, số các số âm, số các số bằng không của dãy các số thực A[n], mỗi số được phân biệt với nhau bởi một hoặc vài dấu trống.
- Ghi lại dãy con dài nhất đã được sắp xếp theo thứ tự tăng dần của dãy các số thực A[n] trên một dòng.
- Ghi lại dãy con dài nhất đã được sắp xếp theo thứ tự giảm dần của dãy các số thực A[n] trên một dòng.
- Ghi lại dãy số đã được sắp xếp theo thứ tự tăng dần của A[n].

1.28. Hãy biểu diễn và xây dựng tập thao tác trên số phức a, b. Bao gồm:

- Tổng hai số phức c = a + b;
- Hiệu hai số phức c = a - b;
- Tích hai số phức c = a * b;
- Thương hai số phức c = a / b;
- Căn bậc hai của số phức.

1.29. Cho tệp dữ liệu text được tổ chức thành từng dòng. Hãy xây dựng tập các thao tác sau trên tệp dữ liệu đó:

- Đếm số dòng trong tệp dữ liệu;
- Đếm số từ và tìm từ dài nhất trong tệp dữ liệu;
- Tìm tần số xuất hiện từ x bất kỳ trong tệp dữ liệu;
- Mã hoá tệp theo kỹ thuật parity;

- Giải mã tệp theo kỹ thuật parity;
- Mã hoá tệp theo kỹ thuật Caresa;
- Giải mã tệp theo kỹ thuật Caresa;

1.30. Bài toán với các số lớn: Biết rằng mỗi số nguyên, dương được biểu diễn như một STRING có độ dài nhỏ hơn 255. Hãy xây dựng các phép toán {+, -, *, Div, Mod} cho các số nguyên đó

CHƯƠNG 2

MẢNG VÀ CON TRỎ (ARRAY AND POINTER)

2.1. CẤU TRÚC LƯU TRỮ MẢNG

2.1.1. Khái niệm về mảng

Mảng là một tập cố định các phần tử cùng có chung một kiểu dữ liệu được lưu trữ kế tiếp nhau trong bộ nhớ. Các thao tác trên mảng bao gồm: tạo lập mảng (create), tìm kiếm một phần tử của mảng (retrieve), lưu trữ mảng (store). Ngoài giá trị, mỗi phần tử của mảng còn được đặc trưng bởi chỉ số của nó (index). Index của một phần tử thể hiện thứ tự của phần tử đó trong mảng. Không có các thao tác bổ sung thêm phần tử hoặc loại bỏ phần tử của mảng vì số phần tử trong mảng là cố định.

Một mảng một chiều gồm n phần tử được coi như một vector n thành phần được đánh số từ 0, 1, 2, ..., n-1. Chúng ta có thể mở rộng khái niệm mảng một chiều cho mảng nhiều chiều như sau:

Một mảng một chiều gồm n phần tử, trong đó mỗi phần tử của nó lại là một mảng một chiều gồm m phần tử được gọi là một mảng hai chiều gồm n×m phần tử.

Tổng quát, một mảng gồm n phần tử mà mỗi phần tử của nó lại là một mảng k - 1 chiều thì nó được gọi là mảng k chiều. Số phần tử của mảng k chiều là tích số giữa số các phần tử của mỗi mảng một chiều.

Khai báo mảng một chiều được thực hiện theo qui tắc như sau:

Tên_kiểu ***Tên_biến[Số_phần_tử];***

Chẳng hạn với khai báo:

```
int A[10];        /* khai báo mảng gồm 10 phần tử nguyên */
```

```
char str[20];      /* khai báo mảng gồm 20 kí tự */
float B[20];       /* khai báo mảng gồm 20 số thực */
long int L[20];    /* khai báo mảng gồm 20 số nguyên dài */
```

2.1.2. Cấu trúc lưu trữ của mảng một chiều

Cấu trúc lưu trữ của mảng: Mảng được tổ chức trong bộ nhớ như một vector, mỗi thành phần của vector được tương ứng với một ô nhớ có kích cỡ đúng bằng kích cỡ của kiểu phần tử và được lưu trữ kế tiếp nhau trong bộ nhớ. Nếu chúng ta có khai báo mảng gồm n phần tử thì phần tử đầu tiên là phần tử thứ 0 và phần tử cuối cùng là phần tử thứ n - 1, đồng thời mảng được cấp phát một vùng không gian nhớ liên tục có số byte được tính theo công thức:

Kích_cỡ_mảng = Số_phần_tử *sizeof (kiểu_phần_tử).

Chẳng hạn trong có khai báo:

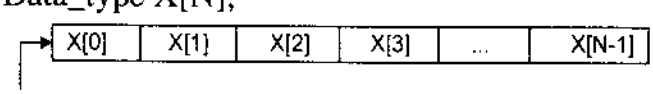
int A[10]; Khi đó kích cỡ tính theo byte của mảng là:

10*sizeof(int) = 20 byte;

float B[20]; khi đó mảng được cấp phát: 20*sizeof(float) = 80byte;

Chương trình dịch của ngôn ngữ C luôn qui định tên của mảng đồng thời là địa chỉ phần tử đầu tiên của mảng trong bộ nhớ. Do vậy, nếu ta có một kiểu dữ liệu nào đó là Data_type, tên của mảng là X, số phần tử của mảng là N thì mảng được tổ chức trong bộ nhớ như sau:

Data_type X[N];



X - là địa chỉ đầu tiên của mảng.

X = &X[0] = (X + 0);

&X[1] = (X + 1);

... ...

&X[i] = (X + i);

Ví dụ 2.1. Kiểm tra cấu trúc lưu trữ của mảng trong bộ nhớ của mảng một chiều.

```
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
#include <io.h>
void main(void) {
        int A[10], i;                   /* khai báo mảng gồm 10 biến nguyên */
        printf("\n Địa chỉ đầu của mảng A là: %p", A);
        printf("\n Kích cỡ của mảng: %5d byte", 10 *sizeof(int));
        for(i = 0; i < 10; i ++) {
                printf("\n Địa chỉ phần tử thứ %5d: %p", i, &A[i]);
        }
        getch();
}
```

Kết quả thực hiện chương trình:

```
Địa chỉ đầu của mảng A là: FFE2
Kích cỡ của mảng: 20
Địa chỉ phần tử thứ 0: FFE2
Địa chỉ phần tử thứ 1: FFE4
Địa chỉ phần tử thứ 2: FFE6
Địa chỉ phần tử thứ 3: FFE8
Địa chỉ phần tử thứ 4: FFEA
Địa chỉ phần tử thứ 5: FFEC
Địa chỉ phần tử thứ 6: FFEE
Địa chỉ phần tử thứ 7: FFF0
Địa chỉ phần tử thứ 8: FFF2
Địa chỉ phần tử thứ 9: FFF4
```

Ví dụ 2.1 in ra địa chỉ của các phần tử trong mảng A gồm 10 phần tử nguyên. Kết quả như được đưa ra ở trên cho ta thấy địa chỉ của mảng trong bộ nhớ trùng với địa chỉ của phần tử A[0] đều bằng FFE2, tiếp đến các phần tử được lưu trữ kế tiếp và cách nhau đúng bằng kích cỡ của kiểu int. Bạn đọc có thể dùng chương trình đơn giản này để kiểm tra cấu trúc lưu trữ của mảng cho các kiểu dữ liệu khác.

2.1.3. Cấu trúc lưu trữ mảng nhiều chiều

Đa số các ngôn ngữ không hạn chế số chiều của mảng, chế độ cấp phát bộ nhớ cho mảng nhiều chiều được thực hiện theo cơ chế ưu tiên theo hàng.

Khai báo mảng nhiều chiều:

Data_type Tên_biến[số_chiều_1] [số_chiều_2] ... [số_chiều_n]

int A[3][3]; khai báo mảng hai chiều gồm 9 phần tử nguyên được lưu trữ liên tục từ A[0][0], A[0][1], A[0][2], A[1][0], A[1][0], A[1][1], A[1][2], A[2][0], A[2][1], A[2][2];

Ví dụ 2.2. Kiểm tra cấu trúc lưu trữ của bảng hai chiều trong bộ nhớ.

```
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
#include <io.h>
void main(void) {
        float A[3][3];      /* khai báo mảng hai chiều gồm 9 phần tử nguyên */
        int i, j;
         /* Địa chỉ của các hàng */
        for(i = 0; i < 3; i++)
                printf("\n Địa chỉ hàng thứ %d: %p", i, A[i]);
        for(i = 0; i < 3; i++) {
                printf("\n");
```

```
            for(j = 0; j < 3; j++)
                    printf("Địa chỉ phần tử %10p:", &A[i][j]);
    }
    getch();
}
```

Kết quả thực hiện chương trình:

```
Địa chỉ hàng thứ 0: FFD2
Địa chỉ hàng thứ 1: FFDE
Địa chỉ hàng thứ 2: FFEA
Địa chỉ phần tử A[0][0]: FFD2
Địa chỉ phần tử A[0][1]: FFD6
Địa chỉ phần tử A[0][2]: FFDA
Địa chỉ phần tử A[1][0]: FFDE
Địa chỉ phần tử A[1][1]: FFE2
Địa chỉ phần tử A[1][2]: FFE6
Địa chỉ phần tử A[2][0]: FFEA
Địa chỉ phần tử A[2][1]: FFEE
Địa chỉ phần tử A[2][2]: FFF2
```

Dễ dàng nhận thấy, địa chỉ hàng thứ i trùng với địa chỉ phần tử đầu tiên trong hàng tương ứng. Tiếp đến các phần tử trong mỗi hàng được lưu trữ cách nhau đúng bằng kích cỡ của kiểu *float*.

Ghi chú: Kết quả thực hiện ví dụ 2.1, 2.2 có thể cho ra kết quả khác nhau trên các máy tính khác nhau, vì việc phân bổ bộ nhớ cho mảng tùy thuộc vào không gian nhớ tự do của mỗi máy.

2.2. CÁC THAO TÁC ĐỐI VỚI MẢNG

Các thao tác đối với mảng bao gồm: tạo lập mảng, tìm kiếm phần tử của mảng, lưu trữ mảng. Các thao tác này có thể được thực hiện

ngay từ khi khai báo mảng. Chúng ta có thể vừa khai báo mảng vừa khởi đầu cho mảng, nhưng cần chú ý một số kỹ thuật khởi đầu cho mảng để vừa đạt được mục đích đề ra vừa tiết kiệm bộ nhớ. Chẳng hạn với khai báo:

int A[10] = {5, 7, 2, 1, 9};

Chương trình vẫn phải cấp phát cho mảng A kích cỡ 10*sizeof(int) = 20 byte bộ nhớ, trong khi đó số byte cần thiết thực sự cho mảng chỉ là 5*sizeof(int) = 10 byte. Để tránh lãng phí bộ nhớ, chúng ta có thể vừa khai báo vừa đồng thời khởi đầu cho mảng như sau.

int A[] = {5, 7, 2, 1, 9};

Với cách khai báo này, miền bộ nhớ cấp phát cho mảng chỉ là số các số nguyên được khởi đầu trong dãy và bằng 5*sizof(int) = 10 byte.

Sau đây là một số ví dụ minh họa cho các thao tác xử lý mảng một và nhiều chiều.

Ví dụ 2.3. Tạo lập mảng các số thực gồm n phần tử, tìm phần tử lớn nhất và chỉ số của phần tử lớn nhất trong mảng.

```
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
#include <io.h>
#define MAX 100                    /* Số phần tử tối đa trong mảng */
void main(void) {
        float A[MAX], max; int i, j, n;
         /* Khởi tạo mảng số */
        printf("\n Nhập số phần tử của mảng n = "); scanf("%d", &n);
        for(i = 0; i < n; i++) {
                printf("\n Nhập A[%d] = ", i); scanf("%f", &A[i]);
```

```
        }
        max = A[0]; j = 0;
        for(i = 1; i < n; i++) {
                if(A[i] > max) {
                        max = A[i]; j = i;
                }
        }
        printf("\n Chỉ số của phần tử lớn nhất là: %d", j);
        printf("\n Giá trị của phần tử lớn nhất là: %6.2f", max);
        getch();
}
```

Kết quả thực hiện chương trình:

```
Nhập số phần tử của mảng n = 7
Nhập A[0] = 1
Nhập A[1] = 9
Nhập A[2] = 2
Nhập A[3] = 8
Nhập A[4] = 3
Nhập A[5] = 7
Nhập A[6] = 4
Chỉ số của phần tử lớn nhất là: 1
Giá trị của phần tử lớn nhất là: 9
```

Ví dụ 2.4. Tạo lập ma trận cấp m x n và tìm phần tử lớn nhất, nhỏ nhất của ma trận.

```
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
#include <io.h>
```

```
#define M 20
#define N 20
void main(void) {
        float A[M][N], max, t; int i, j, k, p, m, n;
        clrscr();
        printf("\n Nhập số hàng của ma trận:"); scanf("%d", &m);
        printf("\n Nhập số cột của ma trận:"); scanf("%d", &n);
        for(i = 0; i < m; i++) {
                for(j = 0; j < n; j++) {
                        printf("\n Nhập A[%d][%d] = ", i, j);
                        scanf("%f", &t); A[i][j] = t;
                }
        }
        max = A[0][0]; k = 0; p = 0;
        for(i = 0; i < m; i++) {
                for(j = 0; j < n; j++) {
                        if(A[i][j] > max) {
                                max = A[i][j]; k = i; p = j;
                                }
                        }
                }
        printf("\n Phần tử có giá trị max là A[%d][%d] = % 6.2f", k, p, max);
        getch();
        }
}
```

Ghi chú: C không hỗ trợ khuôn dạng nhập dữ liệu %f cho các mảng nhiều chiều. Do vậy, muốn nhập dữ liệu là số thực cho mảng nhiều chiều chúng ta phải nhập vào biến trung gian sau đó gán giá trị

trở lại. Đây không phải là hạn chế của C++ do hàm scanf() đã được thay thế bởi toán tử "cin". Tuy nhiên, khi sử dụng cin, cout chúng ta phải viết chương trình dưới dạng *.cpp.

2.3. MẢNG VÀ ĐỐI CỦA HÀM

Như chúng ta đã biết, khi hàm được truyền theo tham biến thì giá trị của biến có thể bị thay đổi sau mỗi lời gọi hàm. Hàm được gọi là truyền theo tham biến khi chúng ta truyền cho hàm là địa chỉ của biến. Ngôn ngữ C qui định tên của mảng đồng thời là địa chỉ của mảng trong bộ nhớ. Do vậy, nếu chúng ta truyền cho hàm là tên của một mảng thì hàm luôn thực hiện theo cơ chế truyền theo tham biến, trường hợp này giống như ta sử dụng từ khoá Var trong khai báo biến của hàm trong Pascal. Trong trường hợp muốn truyền theo tham trị với đối của hàm là một mảng, ta cần phải thực hiện trên một bản sao khác của mảng, khi đó các thao tác đối với mảng thực chất đã được thực hiện trên một vùng nhớ khác dành cho bản sao của mảng.

Ví dụ 2.5. Tạo lập và sắp xếp dãy các số thực A_1, A_2, ..., A_n theo thứ tự tăng dần.

Để giải quyết bài toán, chúng xây dựng chương trình thành 3 hàm riêng biệt: hàm Init_Array() có nhiệm vụ tạo lập mảng số A[n], hàm Sort_Array() thực hiện việc sắp xếp dãy các số được lưu trữ trong mảng, hàm In_Array() in lại kết quả sau khi mảng đã được sắp xếp.

```
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
#include <io.h>
#define MAX 100
/* Khai báo nguyên mẫu cho hàm */
void Init_Array (float A[], int n);
void Sort_Array(float A[], int n);
```

```
void In_Array(float A[], int n);
/* Mô tả hàm */
/* Hàm tạo lập mảng số */
void Init_Array(float A[], int n) {
        int i;
        for(i = 0; i < n; i++) {
                printf("\n Nhập A[%d] = ", i);
                scanf("%f", &A[i]);
        }
}
/* Hàm sắp xếp mảng số */
void Sort_Array(float A[], int n) {
        int i, j; float temp;
        for(i = 0; i < n - 1; i ++) {
                for(j = i + 1; j < n; j ++) {
                        if(A[i] > A[j]) {
                                temp = A[i]; A[i] = A[j]; A[j] = temp;
                        }
                }
        }
}
/* Hàm in mảng số */
void In_Array (float A[], int n) {
        int i;
        for(i = 0; i < n; i++)
                printf("\n Phần tử A[%d] = %6.2f", i, A[i]);
        getch();
}
```

```
/* Chương trình chính */
void main(void) {
        float A[MAX]; int n;
        printf("\n Nhập số phần tử của mảng n = "); scanf("%d", &n);
        Init_Array(A, n);
        Sort_Array(A, n);
        In_Array(A, n);
}
```

Ví dụ 2.6. Viết chương trình tính tổng của hai ma trận cùng cấp.

Chương trình được xây dựng thành 3 hàm, hàm Init_Matrix(): Tạo lập ma trận cấp m x n; hàm Tong_Matrix() tính tổng hai ma trận cùng cấp; hàm In_Matrix() in ma trận kết quả. Tham biến được truyền vào cho hàm là tên ma trận, số hàng, số cột của ma trận.

```
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
#include <io.h>
#include <dos.h>        /* Khai báo sử dụng hàm delay() trong chương trình */
#define M 20            /* Số hàng của ma trận */
#define N 20            /* Số cột của ma trận */
/* Khai báo nguyên mẫu cho hàm */
void Init_Matrix(float A[M][N], int m, int n, char ten);
void Tong_Matrix(float A[M][N], float B[M][N], float C[M][N], int m, int n);
void In_Matrix(float A[M][N], int m, int n);
/* Mô tả hàm */
void Init_Matrix(float A[M][N], int m, int n, char ten) {
        int i, j; float temp; clrscr();
        for(i = 0; i < m; i++) {
```

```
                for(j = 0; j < n; j++) {
                        printf("\n Nhập %c[%d][%d] = ", ten, i, j);
                        scanf("%f", &temp); A[i][j] = temp;
                }
        }
}
void Tong_Matrix(float A[M][N], float B[M][N], float C[M][N], int m, int n) {
        int i, j;
        for(i = 0; i < m; i++) {
                for(j = 0; j < n; j++) {
                        C[i][j] = A[i][j] + B[i][j];
                }
        }
}
void In_Matrix(float A[M][N], int m, int n) {
        int i, j, ch = 179;                     /* 179 là mã kí tự '|' */
        for(i = 0; i < m; i++) {
                printf("\n %-3c", ch);
                for(j = 0; j < n; j++) {
                        printf(" %6.2f", A[i][j];
                }
                printf("%3c", ch);
        }
        getch();
}
/* Chương trình chính */
void main(void) {
        float A[M][N], B[M][N], C[M][N];
```

```
    int n, m; clrscr();
    printf("\n Nhập số hàng m = "); scanf("%d", &m);
    printf("\n Nhập số cột n = "); scanf("%d", &n);
    Init_Matrix(A, m, n, 'A');
    Init_Matrix(B, m, n, 'B');
    Tong_Matrix(A, B, C, m, n);
    In_Matrix(C, m, n);
}
```

2.4. XÂU KÍ TỰ (STRING)

Xâu kí tự là một mảng trong đó mỗi phần tử của nó là một kí tự, kí tự cuối cùng của xâu được dùng làm kí tự kết thúc xâu. Kí tự kết thúc xâu được ngôn ngữ C qui định là kí tự '\0', kí tự này có mã là 0 (NULL) trong bảng mã ASCII. Ví dụ trong khai báo:

char str[] = 'ABCDEF' khi đó xâu kí tự được tổ chức như sau:

0	1	2	3	4	5	6
A	B	C	D	E	F	'\0'

Khi đó str[0] = 'A'; str[1] = 'B', . ., str[5] = 'F', str[6] = '\0';

Vì kí hiệu kết thúc xâu có mã là 0 nên chúng ta có thể kiểm chứng tổ chức lưu trữ của xâu thông qua đoạn chương trình sau:

Ví dụ 2.7. In ra từng kí tự trong xâu.

```
#include <stdio.h>
#include <io.h>
#include <conio.h>
#include <io.h>
#include <dos.h>
#include <string.h>                    /* Sử dụng hàm xử lý xâu kí tự gets() */
```

```
void main(void) {
        char str[20]; int i = 0;
        printf("\n Nhập xâu kí tự:"); gets(str); /* nhập xâu kí tự từ bàn phím */
        while(str[i]!= '\0') {
                putch(c); i++;
        }
}
```

Ghi chú: Hàm getch() nhận một kí tự từ bàn phím, hàm putch(c) đưa ra màn hình kí tự c. Hàm scanf("%s", str): nhận một xâu kí tự từ bàn phím nhưng không được chứa kí tự trống (space), hàm gets(str): cho phép nhận từ bàn phím một xâu kí tự kể cả dấu trống.

Ngôn ngữ C không cung cấp các phép toán trên xâu kí tự, mà mọi thao tác trên xâu kí tự đều phải được thực hiện thông qua các lời gọi hàm. Sau đây là một số hàm xử lý xâu kí tự thông dụng được khai báo trong tệp string.h:

puts (string): Đưa ra màn hình một string.

gets(string): Nhận từ bàn phím một string.

scanf("%s", string): Nhận từ bàn phím một string không kể kí tự trống (space).

strlen(string): Hàm trả lại một số là độ dài của xâu string.

strcpy(s, p): Hàm copy xâu p vào xâu s.

strcat(s, p): Hàm nối xâu p vào sau xâu s.

strcmp(s, p): Hàm trả lại giá trị dương nếu xâu s lớn hơn xâu p, trả lại giá trị âm nếu xâu s nhỏ hơn xâu p, trả lại giá trị 0 nếu xâu s đúng bằng xâu p.

strstr(s, p): Hàm trả lại vị trí của xâu p trong xâu s, nếu p không có mặt trong s hàm trả lại con trỏ NULL.

strncmp(s, p, n): Hàm so sánh n kí tự đầu tiên của xâu s và p.

strncpy(s, p, n): Hàm copy n kí tự đầu tiên từ xâu p vào xâu s.

strrev(str): Hàm đảo xâu s theo thứ tự ngược lại.

Chúng ta có thể sử dụng trực tiếp các hàm xử lý xâu kí tự bằng việc khai báo chỉ thị #include <string.h>. Tuy nhiên, chúng ta có thể viết lại các thao tác đó thông qua ví dụ sau:

Ví dụ 2.8. Xây dựng các thao tác sau cho string.

F1 - Nhập xâu kí tự từ bàn phím hàm gets(str).

F2 - Tìm độ dài xâu kí tự strlen(str).

F3 - Tìm vị trí kí tự C đầu tiên xuất hiện trong xâu kí tự.

F4 - Đảo ngược xâu kí tự.

F5 - Đổi xâu kí tự từ in thường thành in hoa.

F6 - Sắp xếp xâu kí tự theo thứ tự tăng dần.

```
/* Các thao tác với xâu kí tự */
#include <stdio.h>
#include <io.h>
#include <conio.h>
#include <io.h>
#include <dos.h>
#include <string.h>
#define F1      59
#define F2      60
#define F3      61
#define F4      62
#define F5      63
#define F6      64
#define F7      65
```

```
#define F10     68
#define MAX     256
/* Khai báo nguyên mẫu cho hàm */
char *gets (char str[]);            /* char *được hiểu là một xâu kí tự */
int strlen(char str[]);             /* hàm trả lại độ dài xâu */
int strcstr(char str[], char c);    /* hàm trả lại vị trí kí tự c đầu tiên trong str */
char *strrev(char str[]);           /* hàm đảo xâu str */
char *upper(char str[]);            /* hàm đổi xâu str thành chữ in hoa */
char *sort_str(char str[]);         /* hàm sắp xếp xâu theo thứ tự từ điển */
void thuc_hien(void);
/* Mô tả hàm */
/* Hàm trả lại một xâu kí tự được nhập từ bàn phím */
char *gets(char str[]) {
        int i = 0; char c;
        while((c = getch())!= '\n') {       /* nhập nếu không phải phím enter */
                str[i] = c; i++;
        }
        str[i] = '\0';
        return(str);
}
/* Hàm tính độ dài xâu kí tự */
int strlen(char str[]) {
        int i = 0;
        while(str[i]) i++;
        return(i);
}
/* Hàm trả lại vị trí đầu tiên kí tự c trong xâu str */
int strcstr (char str[], char c) {
```

```
        int i = 0;
        while(str[i] && str[i] != c)
                i++;
        if(str[i] = '\0') return(-1);
                return(i);
}
/* Hàm đảo xâu kí tự */
char *strrev(char str[]) {
        int i, j, n = strlen(str); char c;
        i = 0; j = n-1;
        while(i < j) {
                c = str[i]; str[i] = str [j]; str[j] = c;
        }
        return(str);
}
/* Hàm đổi xâu in thường thành in hoa */
char *upper(char str[]) {
        int i, n = strlen(str);
        for(i = 0; i < n; i++) {
                if(str[i] > = 'a' && str[i] < = 'z')
                        str[i] = str[i] - 32;
        }
        return(str);
}
/* Hàm sắp xếp xâu kí tự */
char *sort_str(char str[]) {
        int i, j, n = strlen(str); char temp;
        for(i = 0; i < n-1; i++) {
```

```
                for(j = 0; j < n; j ++) {
                        if(str[i] > str[j]) {
                                temp = str[i]; str[i] = str[j];
                                str[j] = temp;
                        }
                }
        }
}
/* Hàm thực hiện chức năng */
void thuc_hien(void) {
        char c, phim, str[MAX]; int control = 0;
        textmode(0);
        do {
                clrscr();
                printf("\n TẬP THAO TÁC VỚI STRING");
                printf("\n F1 - Tạo lập string");
                printf("\n F2 - Tính độ dài xâu");
                printf("\n F3 - Tìm kí tự trong string");
                printf("\n F4 - Đảo ngược string");
                printf("\n F5 - Đổi thành in hoa");
                printf("\n F6 - Sắp xếp string");
                printf("\n F10 - Trở về");
                phim = getch();
                switch(phim) {
                        case F1: gets(str); control = 1; break;
                        case F2:
                                if(control)
                                        printf("\n Độ dài xâu là:%d", strlen(str));
                                break;
```

```
                    case F3:
                        if(control) {
                            printf("\n Kí tự cần tìm:");
                            scanf("%c", &c);
                            if(strcstr(str, c) >= 0)
                                printf("\n Vị trí:%d", strcstr(str, c));
                        }
                        break;
                    case F4:
                        if(control)
                            printf("\n Xâu đảo:%s", strrev(str));
                        break;
                    case F5:
                        if(control)
                            printf("\n In hoa:%s", upper(str));
                        break;
                    case F6:
                        if(control)
                            printf("\n Xâu được sắp xếp:%s", sort_str(str));
                        break;
                }
                delay(2000);
        } while(phim != F10);
}
/* Chương trình chính */
void main(void) {
                thuc_hien();
}
```

Mảng các string: mảng các string là một mảng mà mỗi phần tử của nó là một string. Chẳng hạn trong khai báo:

```
char buffer[25][80];
```

Đây là mảng các string gồm 25 hàng trong đó mỗi hàng gồm 80 kí tự. Ví dụ sau đây sẽ minh họa cho các thao tác trên mảng các string.

Ví dụ 2.9. Hãy tạo lập mảng các string trong đó mỗi string là một từ khoá của ngôn ngữ lập trình C. Sắp xếp mảng các từ khoá theo thứ tự từ điển.

Chương trình sau sẽ được thiết kế thành 3 hàm chính:

Hàm Init_KeyWord(): thiết lập bảng từ khoá, hàm Sort_KeyWord(): sắp xếp mảng từ khoá; hàm In_KeyWord(): in mảng các từ khoá. Chương trình được thực hiện như sau:

```
/* Thao tác với mảng các string */
#include <stdio.h>
#include <stdlib.h>
#include <io.h>
#include <conio.h>
#include <dos.h>
#include <string.h>
#include <alloc.h>
 /* Khai báo nguyên mẫu cho hàm */
void Init_KeyWord(char key_word[][20], int n);
void Sort_KeyWord(char key_word[][20], int n);
void In_KeyWord(char key_word[][20], int n);
/* Mô tả hàm */
void Init_KeyWord(char key_word[][20], int n) {
        int i;
        for(i = 0; i < n; i++) {
                printf("\n Nhập từ khoá %d:", i);
```

```
                scanf("%s", key_word[i]);
        }
}
void Sort_KeyWord(char key_word[][20], int n) {
        int i, j; char temp[20];
        for(i = 0; i < n - 1; i++) {
                for(j = i +1; j < n; j++) {
                        if(strcmp(key_word[i], key_word[j]) > 0) {
                                strcpy(temp, key_word[i]);
                                strcpy(key_word[i], key_word[j]);
                                strcpy(key_word[j], temp);
                        }
                }
        }
}
void In_KeyWord(char key_word[][20], int n) {
        int i;
        for(i = 0; i < n; i++) {
                printf("\n Key_Word[%d] = %s", i, key_word[i]);
        }
        getch();
}
void main(void) {
        char key_word[100][20]; int n;
        printf("\n Nhập số từ khoá n = "); scanf("%d", &n);
        Init_KeyWord(key_word, n);
        Sort_KeyWord(key_word, n);
        In_KeyWord(key_word, n);
}
```

2.5. CON TRỎ (POINTER)

Con trỏ là biến chứa địa chỉ của một biến khác. Con trỏ được sử dụng rất nhiều trong C và được coi là thế mạnh trong biểu diễn tính toán và truy nhập gián tiếp các đối tượng.

2.5.1. Các phép toán trên con trỏ

Để khai báo con trỏ, chúng ta thực hiện theo cú pháp:

Kiểu_dữ_liệu *Biến_con_trỏ;

Vì con trỏ chứa địa chỉ của đối tượng nên có thể thâm nhập vào đối tượng "gián tiếp" thông qua con trỏ. Giả sử x là một biến kiểu int và px là con trỏ được khai báo:

int x, *px;

Phép toán một ngôi & cho địa chỉ của đối tượng cho nên câu lệnh

px = &x;

Sẽ gán địa chỉ của x cho biến px; px bây giờ được gọi là "trỏ tới" x. Phép toán & chỉ áp dụng được cho các biến và phần tử mảng; kết cấu kiểu &(x + 1) và &3 là không hợp lệ.

Phép toán một ngôi * coi đối tượng của nó là địa chỉ cần xét và thâm nhập tới địa chỉ đó để lấy ra nội dung của biến. Ví dụ, nếu y là int thì:

y = *px;

Sẽ gán cho y nội dung của biến mà px trỏ tới. Vậy dãy:

px = &x;

y = *px;

Sẽ gán giá trị của x cho y như trong lệnh:

y = x;

Cũng cần phải khai báo cho các biến tham dự vào việc này:

int x, y;

int *px;

Khai báo của x và y là điều ta đã biết. Khai báo của con trỏ px có điểm mới:

int *px;

Có ngụ ý rằng tổ hợp *px có kiểu int. Con trỏ có thể xuất hiện trong các biểu thức. Chẳng hạn, nếu px trỏ tới số nguyên x thì *px có thể xuất hiện trong bất kì ngữ cảnh nào mà x có thể xuất hiện.

y = *px + 1;	sẽ đặt y lớn hơn x 1 đơn vị;
printf("%d \ n", *px);	in ra giá trị hiện tại của x.

Phép toán một ngôi * và & có mức ưu tiên cao hơn các phép toán số học, cho nên biểu thức này lấy bất kì giá trị nào mà px trỏ tới, cộng với 1 rồi gán cho y.

Con trỏ cũng có thể xuất hiện bên vế trái của phép gán. Nếu px trỏ tới x thì ***px = 0**; sẽ đặt x thành không và ***px += 1**; sẽ tăng x lên như trong trường hợp (*px) ++;

Các dấu ngoặc là cần thiết trong ví dụ cuối; nếu không có chúng thì biểu thức sẽ tăng px thay cho việc tăng chỗ nó trỏ tới, bởi vì phép toán một ngôi như *và ++ được tính từ phải sang trái.

Cuối cùng, vì con trỏ là biến nên ta có thể thao tác chúng như đối với các biến khác. Nếu py là con trỏ nữa kiểu int, thì:

py = px; sẽ sao nội dung của px vào py, nghĩa là làm cho py trỏ tới nơi mà px trỏ. Ví dụ sau minh họa những thao tác truy nhập gián tiếp tới biến thông qua con trỏ.

Ví dụ 2.10. Thay đổi nội dung của hai biến a và b thông qua con trỏ.

```
#include <stdio.h>
#include <io.h>
```

```
#include <conio.h>
#include <io.h>
#include <dos.h>
#include <string.h>
void main(void) {
        int a = 5, b = 7;           /* Giả sử có hai biến nguyên a = 5, b = 7 */
        int *px, *py;               /* Khai báo hai con trỏ kiểu int */
        px = &a;                    /* px trỏ tới x */
        printf("\n Nội dung con trỏ px = %d", *px);
        *px = *px + 10;             /* Nội dung của *px là 15 */
        /* Con trỏ px đã thay đổi nội dung của a */
        printf("\n Giá trị của a = %d", a);
        px = &b;                    /* px trỏ tới b */
        py = px;
        /* Con trỏ py thay đổi giá trị của b thông qua con trỏ px */
        *py = *py + 10;
        printf("\n Giá trị của b = %d", b);
}
```

Kết quả thực hiện chương trình:

```
Nội dung con trỏ px = 5
Giá trị của a = 15
Giá trị của b = 17
```

2.5.2. Con trỏ và đối của hàm

Để thay đổi trực tiếp nội dung của biến trong hàm thì đối của hàm phải là một con trỏ. Đối với những biến có kiểu cơ bản, chúng ta sử dụng toán tử &(tên_biến) để truyền địa chỉ của biến cho hàm như trong ví dụ đổi nội dung của biến x và biến y trong hàm swap(&x, &y) sau:

```
void swap(int *px, int *py) {
        int temp;
        temp = *px; *px = *py; *py = temp;
}
```

Con trỏ thông thường được sử dụng trong các hàm phải cho nhiều hơn một giá trị (có thể nói rằng swap cho hai giá trị, các giá trị mới thông qua đối của nó).

2.5.3. Con trỏ và mảng

Mảng trong C thực chất là một hằng con trỏ, do vậy, mọi thao tác đối với mảng đều có thể được thực hiện thông qua con trỏ. Khai báo

int a[10];

Xác định mảng có kích thước 10 phần tử int, tức là một khối 10 đối tượng liên tiếp a[0], a[1], ..., a[9]. Cú pháp a[i] có nghĩa là phần tử của mảng ở vị trí thứ i kể từ vị trí đầu. Nếu pa là con trỏ tới một số nguyên, được khai báo là:

int *pa;

Thì phép gán:

pa = &a[0];

Sẽ đặt pa để trỏ tới phần tử đầu của a; tức là pa chứa địa chỉ của a[0]. Bây giờ phép gán:

x = *pa;

Sẽ sao nội dung của a[0] vào x.

Nếu pa trỏ tới một phần tử của mảng a thì theo định nghĩa pa + 1 sẽ trỏ tới phần tử tiếp theo và pa - i sẽ trỏ tới phần tử thứ i trước pa, pa + i sẽ trỏ tới phần tử thứ i sau pa. Vậy nếu pa trỏ tới a[0] thì:

***(pa + 1)**

Sẽ cho nội dung của a[1], pa+i là địa chỉ của a[i] còn *(pa + i) là nội dung của a[i].

Định nghĩa "cộng 1 vào con trỏ" và mở rộng cho mọi phép toán số học trên con trỏ được tăng theo tỉ lệ kích thước lưu trữ của đối tượng mà pa trỏ tới. Vậy trong pa + i, i sẽ được nhân với kích thước của kiểu dữ liệu mà pa trỏ tới trước khi được cộng vào pa.

Sự tương ứng giữa việc định chỉ số và phép toán số học trên con trỏ là rất chặt chẽ. Trong thực tế, việc tham trỏ tới mảng được trình biên dịch chuyển thành con trỏ tới điểm đầu của mảng. Kết quả là tên mảng chính là một biểu thức con trỏ. Vì tên mảng là đồng nghĩa với việc định vị phần tử thứ 0 của mảng a nên phép gán:

pa = &a[0];

Cũng có thể viết là:

pa = a;

Điều cần chú ý là để tham trỏ tới a[i] có thể được viết dưới dạng *(a + i). Trong khi xử lý a[i], C chuyển tức khắc nó thành *(a + i); hai dạng này hoàn toàn là tương đương nhau. Áp dụng phép toán & cho cả hai dạng này ta có &a[i] và (a + i) là địa chỉ của phần tử thứ i trong mảng a. Mặt khác, nếu pa là con trỏ thì các biểu thức có thể sử dụng nó kèm thêm chỉ số: pa[i] đồng nhất với *(pa + i). Tóm lại, bất kì một mảng và biểu thức chỉ số nào cũng đều được viết như một con trỏ.

Có một sự khác biệt giữa tên mảng và con trỏ cần phải nhớ đó là: Con trỏ là một biến, nên pa = a và pa++ đều là các phép toán đúng, còn tên mảng là một hằng chứ không phải là biến nên kết cấu kiểu a = pa hoặc a++ hoặc p = &a là không hợp lệ.

Khi truyền một tên mảng cho hàm thì tên của mảng là địa chỉ đầu của mảng, do vậy, tên mảng thực sự là con trỏ. Ta có thể dùng sự kiện này để viết ra bản mới của strlen, tính chiều dài của xâu kí tự.

```
int strlen(char *s) {                /* Cho chiều dài của xâu s */
        int n;
        for(n = 0; *s != '\ 0'; s ++)
                n ++;
        return(n);
}
```

Việc tăng s là hoàn toàn hợp lệ vì s là biến con trỏ; s++ không ảnh hưởng tới xâu kí tự trong hàm gọi tới strlen mà chỉ làm tăng bản sao riêng của địa chỉ trong strlen.

Vì các tham biến hình thức trong định nghĩa hàm

char s[ ]; và **char** *s là hoàn toàn tương đương. Khi truyền một tên mảng cho hàm, hàm có thể coi rằng nó xử lí hoặc mảng hoặc con trỏ là giống nhau. Nếu p và q trỏ tới các thành phần của cùng một mảng thì có thể áp dụng được các quan hệ như <, <=, >= v.v... chẳng hạn:

p < q

Là đúng, nếu p con trỏ tới thành phần đứng trước thành phần mà q trỏ tới trong mảng. Các quan hệ = = và != cũng áp dụng được. Có thể so sánh một con trỏ với giá trị NULL. Nhưng so sánh các con trỏ trỏ tới các mảng khác nhau sẽ không đưa lại kết quả mong muốn. Tiếp nữa, con trỏ và số nguyên có thể cộng hoặc trừ cho nhau như kết cấu:

p + n

Nghĩa là đối tượng thứ n sau đối tượng do p trỏ tới. Điều này đúng với bất kể loại đối tượng nào mà p được khai báo sẽ trỏ tới; trình biên dịch sẽ tính tỉ lệ n theo kích thước của các đối tượng do p trỏ tới, điều này được xác định theo khai báo của p. Chẳng hạn trên máy tính PC AT, nhân từ tỉ lệ là 1 đối với char, 2 đối với int và short, 4 cho long và float.

Phép trừ con trỏ cũng hợp lệ; nếu p và q trỏ tới các thành phần của cùng một mảng thì p - q là số phần tử giữa p và q. Dùng sự kiện này ta có thể viết ra một bản khác cho strlen:

```
/* Cho độ dài xâu s */
int strlen(char *s) {
        char *p = s;
        while(*p != '\0')
                p ++;
        return(p-s);
}
```

Trong khai báo, p được khởi đầu là s, tức là trỏ tới kí tự đầu tiên. Chu trình while, kiểm tra lần lượt từng kí tự xem đã là '\0' chưa để xác định cuối xâu. Vì '\0' là 0 và vì while chỉ kiểm tra xem biểu thức có là 0 hay không nên có thể bỏ phép thử tường minh. Vậy ta có thể viết lại chu trình trên:

```
while(*p)
        p ++;
```

Do p trỏ tới các kí tự nên p ++ sẽ chuyển p để trỏ sang kí tự tiếp theo và p - s sẽ cho số các kí tự đã lướt qua, tức là độ dài của xâu. Ví dụ sau minh họa rõ hơn về phương pháp sử dụng con trỏ.

Ví dụ 2.11. Viết chương trình sắp xếp dãy các số nguyên theo thứ tự tăng dần bằng con trỏ.

Chương trình được thiết kế thành 3 hàm chính, hàm Init_Array() tạo lập dãy các số thực, hàm Sort_Array() sắp xếp dãy các số thực theo thứ tự tăng dần, hàm In_Array() in kết quả đã được sắp xếp.

```
#include <stdio.h>
#include <io.h>
#include <conio.h>
#include <io.h>
#include <dos.h>
#include <string.h>
#define MAX 100
```

```
/* Khai báo nguyên mẫu cho hàm */
void Init_Array(float *A, int n);
void Sort_Array(float *A, int n);
void In_Array(float *A, int n);
/* Mô tả hàm */
void Init_Array(float *A, int n) {
        int i;
        for(i = 0; i < n; i++) {
                printf("\n Nhập A[%d] = ", i);
                scanf("%f", (A + i));
        }
}
void Sort_Array(float *A, int n) {
        int i, j; float temp;
        for(i = 0; i < n -1; i++) {
                for(j = i +1; j < n; j++) {
                        if(*(A+i) > *(A + j)) {
                                temp = *(A +i);
                                *(A + i) = *(A + j);
                                *(A +j) = temp;
                        }
                }
        }
}
void In_Array(float *A, int n) {
        int i;
        for(i = 0; i < n; i++) {
                printf("\n Phần tử A[%d] = % 6.2f", i, *(A +i));
```

```
        }
}
/* Chương trình chính */
void main(void) {
        float A[MAX]; int n;
        printf("\n Nhập n = "); scanf("%d", &n);
        Init_Array(A, n);
        Sort_Array(A, n);
        In_Array(A, n);
}
```

Ví dụ 2.11. Xây dựng các thao tác tạo lập đa thức, tính giá trị đa thức theo lược đồ Hooneir, tính tổng hai đa thức, hiệu hai đa thức.

Chương trình được xây dựng thành các hàm sau:

Hàm Init_Dathuc(): Tạo lập đa thức.

Hàm Val_Dathuc(): Tính giá trị của đa thức tại điểm X_0

Hàm In_Dathuc(): In dưới dạng hệ số của đa thức

Hàm Tong_Dathuc(): Tính tổng hai đa thức

Hàm Hieu_Dathuc(): Tính hiệu hai đa thức.

Hàm Thuc_Hien(): Điều khiển chương trình bằng menu

```
/* Chương trình xây dựng tập thao tác cho đa thức */
#include <stdio.h>
#include <io.h>
#include <conio.h>
#include <io.h>
#include <dos.h>
#include <string.h>
#define MAX 100
#define F1      59
```

```
#define F2      60
#define F3      61
#define F4      62
#define F5      63
#define F10     68
/* Nguyên mẫu của hàm */
void Init_Dathuc(float *A, int n, char c);
void In_Dathuc(float *A, int n);
float Val_Dathuc(float *A, int n);
void Tong_Dathuc(float *C, float *A, float *B, int n, int m);
void Hieu_Dathuc(float *C, float *A, float *B, int n, int m);
void Thuc_Hien(void);
/* Mô tả hàm */
void Init_Dathuc(float *A, int n, char c) {
        int i; float t;
        for(i = 0; i < n; i++) {
                printf("\n %c[%d] = ", c, i);
                scanf("%f", &t);
                A[i] = t;
        }
        In_Dathuc(A, n);
}
void In_Dathuc(float *A, int n) {
        int i;
        for(i = 0; i < n; i++)
                printf("%7.2f", A[i]);
        printf("\n");
        delay(2000);
}
```

```
float Val_Dathuc(float *A, int n) {
        float s = A[n-1], k; int i, j;
        printf("\n Nhập k = "); scanf("%f", &k);
        for(i = n-2; i >= 0; i--)
                s = s*k + A[i];
        printf("\n S = %6.2f", s); delay(2000);
        return(s);
}
void Tong_Dathuc(float *C, float *A, float *B, int n, int m) {
        int k, i, j;
        if(n > m) {
                k = n;
                for(i = 0; i < m; i++)
                        C[i] = A[i]+B[i];
                for(i = m; i < n; i++)
                        C[i] = A[i];
        }
        else {
                k = m;
                for(i = 0; i < n; i++)
                        C[i] = A[i]+B[i];
                for(i = n; i < m; i++)
                        C[i] = B[i];
        }
        printf("\n Đa thức tổng:");
        In_Dathuc(C, k);
}
void Hieu_Dathuc(float *C, float *A, float *B, int n, int m) {
```

```
        int k, i, j;
        if(n > m) {
                k = n;
                for(i = 0; i < m; i++)
                        C[i] = A[i]-B[i];
                for(i = m; i < n; i++)
                        C[i] = A[i];
        }
        else {
                k = m;
                for(i = 0; i < n; i++)
                        C[i] = A[i]-B[i];
                for(i = n; i < m; i++)
                        C[i] = B[i];
        }
        printf("\n Đa thức hiệu:");
        In_Dathuc(C, k);
}
void Thuc_Hien(void) {
        float A[MAX], B[MAX], C[MAX];
        int n, m, control = 0; char phim;
        printf("\n Bậc của đa thức A:"); scanf("%d", &n);
        printf("\n Bậc của đa thức B:"); scanf("%d", &m);
        n = n+1; m = m+1;
        do {
                clrscr();
                printf("\n TẬP THAO TÁC TRÊN ĐA THỨC");
                printf("\n F1 - Tạo lập đa thức");
```

```
            printf("\n F2 - Tính giá trị đa thức");
            printf("\n F3 - Tổng hai đa thức");
            printf("\n F4 - Hiệu hai đa thức");
            printf("\n F10 - Tạo lập đa thức");
            phim = getch();
            switch(phim) {
                case F1: control = 1;
                    Init_Dathuc(A, n, 'A');
                    Init_Dathuc(B, m, 'B'); break;
                case F2:
                    if(control) {
                        Val_Dathuc(A, n);
                        Val_Dathuc(B, m);
                    }
                    break;
                case F3:
                    if(control)
                        Tong_Dathuc(C, A, B, n, m);
                    break;
                case F4:
                    if(control)
                        Hieu_Dathuc(C, A, B, n, m);
                    break;
            }
        } while(phim != F10);
}
void main(void) {
        Thuc_Hien();
}
```

2.5.4. Cấp phát bộ nhớ cho con trỏ

Nếu p là con trỏ thì p++ sẽ tăng p để trỏ tới phần tiếp của đối tượng mà p đã trỏ tới, còn p += i sẽ tăng lên trỏ tới phần tử thứ i kể từ sau phần tử p trỏ tới. Đa số các trường hợp chúng ta đã xét thì p được trỏ tới một mảng nào đó đã được khai báo trước, khi đó mặc nhiên vùng bộ nhớ mà con trỏ đang xử lý là vùng bộ nhớ đã được cấp phát cho mảng trước đó và ta có thể xử lý như một mảng. Tuy nhiên, chúng ta có thể sử dụng con trỏ linh động hơn bằng các phương pháp cấp phát bộ nhớ động cho con trỏ.

Trong thư viện chuẩn của C cung cấp cho ta một số hàm cấp phát bộ nhớ cho con trỏ.

Hàm malloc(size_t size): cấp phát một vùng bộ nhớ có kích cỡ size cho con trỏ. Hàm trả lại con trỏ NULL nếu size = 0;

Hàm calloc(n, i): đặt con trỏ vào đầu vùng bộ nhớ gồm n x i bytes tự do. Hàm trả lại con trỏ NULL nếu việc cấp phát không thành công.

Hàm farmalloc(), farcalloc(): đặt con trỏ vào đầu vùng bộ nhớ tự do ngoài vùng heap, trả lại con trỏ NULL nếu việc cấp phát không thành công.

Hàm free (p): giải phóng vùng bộ nhớ đã cấp phát bởi malloc() hoặc calloc() trước đó.

Hàm farfree(): giải phóng vùng bộ nhớ đã cấp phát bởi farmalloc() hoặc farcalloc() trước đó.

Trong ví dụ 2.11, khi xây dựng tập các thao tác cho đa thức, chúng ta dùng khai báo:

```
float A[MAX], B[MAX], C[MAX];
```

MAX được định nghĩa để qui định bậc lớn nhất của đa thức có thể có, điều này làm hạn chế đến phạm vi của bài toán, chính xác hơn chương trình chỉ giải quyết lớp các đa thức có bậc nhỏ hơn MAX.

Nhưng MAX là bao nhiêu thì được coi là bậc lớn nhất của đa thức, nếu MAX có giá trị lớn trong khi đó các thao tác chỉ xử lý những đa thức có bậc nhỏ sẽ gây nên lãng phí bộ nhớ, nếu khắc phục tình trạng này bằng việc thu nhỏ MAX thì với những thao tác cho đa thức có bậc lớn lại không thể xử lý được. Để khắc phục tình trạng này, chúng ta có thể sử dụng con trỏ với phương thức cấp phát bộ nhớ hoàn toàn động. Sau đây là một version mới xây dựng tập thao tác cho đa thức bằng con trỏ.

Ví dụ 2.12. Xây dựng các thao tác tạo lập đa thức, tính giá trị đa thức theo lược đồ Hooneir, tính tổng hai đa thức, hiệu hai đa thức.

```
/* Chương trình xây dựng tập thao tác cho đa thức */
#include <stdio.h>
#include <stdlib.h>
#include <io.h>
#include <conio.h>
#include <dos.h>
#include <string.h>
#include <alloc.h>
#define F1      59
#define F2      60
#define F3      61
#define F4      62
#define F5      63
#define F10     68
/* Nguyên mẫu của hàm */
void Init_Dathuc(float *A, int n, char c);
void In_Dathuc(float *A, int n);
float Val_Dathuc(float *A, int n);
void Tong_Dathuc(float *C, float *A, float *B, int n, int m);
void Hieu_Dathuc(float *C, float *A, float *B, int n, int m);
```

```
void Thuc_Hien(void);
/* Mô tả hàm */
void Init_Dathuc(float *A, int n, char c) {
        int i; float t;
        for(i = 0; i < n; i++) {
                printf("\n %c[%d] = ", c, i);
                scanf("%f", &t);
                A[i] = t;
        }
        In_Dathuc(A, n);
}
void In_Dathuc(float *A, int n) {
        int i;
        for(i = 0; i < n; i++)
                printf("%7.2f", A[i]);
        printf("\n");
        delay(2000);
}
float Val_Dathuc(float *A, int n) {
        float s = A[n-1], k; int i, j;
        printf("\n Nhập k = "); scanf("%f", &k);
        for(i = n-2; i > = 0; i--)
                s = s*k + A[i];
        printf("\n S = %6.2f", s); delay(2000);
        return(s);
}
void Tong_Dathuc(float *C, float *A, float *B, int n, int m) {
        int k, i, j;
```

```
        if(n > m) {
                k = n;
                for(i = 0; i < m; i++)
                        C[i] = A[i]+B[i];
                for(i = m; i < n; i++)
                        C[i] = A[i];
        }
        else {
                k = m;
                for(i = 0; i < n; i++)
                        C[i] = A[i]+B[i];
                for(i = n; i < m; i++)
                        C[i] = B[i];
        }
        printf("\n Đa thức tổng:");
        In_Dathuc(C, k);
}
void Hieu_Dathuc(float *C, float *A, float *B, int n, int m) {
        int k, i, j;
        if(n > m) {
                k = n;
                for(i = 0; i < m; i++)
                        C[i] = A[i]-B[i];
                for(i = m; i < n; i++)
                        C[i] = A[i];
        }
        else {
                k = m;
```

```
            for(i = 0; i < n; i++)
                    C[i] = A[i]-B[i];
            for(i = n; i < m; i++)
                    C[i] = B[i];
        }
        printf("\n Đa thức hiệu:");
        In_Dathuc(C, k);
}
void Thuc_Hien(void) {
        float *A, *B, *C;
        int n, m, control = 0; char phim;
        printf("\n Bac cua da thuc A:"); scanf("%d", &n);
        printf("\n Bac cua da thuc B:"); scanf("%d", &m);
        n = n+1; m = m+1;
        A = (float *) malloc(n *szeof(float));
        B = (float *) malloc(m *szeof(float));
        C = (float *) malloc((m + n) *szeof(float));
        do {
            clrscr();
            printf("\n TẬP THAO TÁC TRÊN ĐA THỨC");
            printf("\n F1 - Tạo lập đa thức");
            printf("\n F2 - Tính giá trị đa thức");
            printf("\n F3 - Tổng hai đa thức");
            printf("\n F4 - Hiệu hai đa thức");
            printf("\n F10 - Tạo lập đa thức");
            phim = getch();
            switch(phim) {
                    case F1: control = 1;
```

```
                        Init_Dathuc(A, n, 'A');
                        Init_Dathuc(B, m, 'B'); break;
                case F2:
                        if(control) {
                                Val_Dathuc(A, n);
                                Val_Dathuc(B, m);
                        }
                        break;
                case F3:
                        if(control)
                                Tong_Dathuc(C, A, B, n, m);
                        break;
                case F4:
                        if(control)
                                Hieu_Dathuc(C, A, B, n, m);
                        break;
            }
        } while(phim != F10);
        free(A);        free(B);        free(C);
}
void main(void) {
        Thuc_Hien();
}
```

2.6. CON TRỎ VỚI MẢNG NHIỀU CHIỀU

Ngôn ngữ C không hạn chế số chiều của mảng nhiều chiều mặc dù trong thực tế có khuynh hướng sử dụng mảng con trỏ nhiều hơn. Trong mục này, ta sẽ đưa ra mối liên hệ giữa con trỏ và mảng nhiều chiều.

Khi xử lý với các mảng nhiều chiều, thông thường lập trình viên thường đưa ra khai báo float A[3][3]; hoặc float A[N][N] với N được định nghĩa là cực đại của cấp ma trận vuông mà chúng ta cần giải quyết. Để tạo lập ma trận, chúng ta cũng có thể xây dựng thành hàm riêng:

```
Init_Matrix(float A[N][N], int n);
```

Hoặc có thể khởi đầu trực tiếp ngay sau khi định nghĩa:

```
float A[3][3] = {
                {1, 2, 4},
                {4, 8, 12},
                {3, -3, 0}
        }
```

```
Hoặc A[][3] = {
                {1, 2, 4},
                {4, 8, 12},
                {3, -3, 0}
        }
```

Cả hai cách khởi đầu đều bắt buộc phải có chỉ số cột. Nhưng để thấy rõ được mối liên hệ giữa mảng nhiều chiều và con trỏ, chúng ta phải phân tích kỹ cấu trúc lưu trữ của mảng nhiều chiều.

Vì tên của mảng là địa chỉ của mảng đó trong bộ nhớ điều đó dẫn tới những kết quả như sau:

Địa chỉ của ma trận A[3][3] là A đồng thời là địa chỉ hàng thứ 0 đồng thời là địa chỉ của phần tử đầu tiên của ma trận:

Địa chỉ đầu của ma trận A là A = A[0] = *(A + 0) = & A[0][0];

Địa chỉ hàng thứ nhất: A[1] = *(A + 1) = &A[1][0];

Địa chỉ hàng thứ i: A[i] = *(A+i) = &A[i][0];

Địa chỉ phần tử A[i][j] = (*(A+i)) + j;

Nội dung phần tử A[i][j] = *(*(A+i)) + j);

Ví dụ 2.13. Kiểm chứng lại mối quan hệ giữa mảng nhiều chiều với con trỏ thông qua cấu trúc lưu trữ của nó trong bộ nhớ.

```
#include <stdio.h>
#include <stdlib.h>
#include <io.h>
#include <conio.h>
#include <dos.h>
#include <string.h>
#include <alloc.h>
void main(void) {
        float A[3][3];      /* Khai báo mảng hai chiều gồm 9 phần tử nguyên */
        int i, j;
        /* Địa chỉ của các hàng */
        for(i = 0; i < 3; i++)
                printf("\n Địa chỉ hàng thứ %d là:%p", i, A[i]);
        /* Địa chỉ hàng truy nhập thông qua con trỏ */
        printf("\n Truy nhập bằng con trỏ:");
        for(i = 0; i < 3; i++)
                printf("\n Địa chỉ hàng thứ %d là:%p", i, *(A+i));
        /* Địa chỉ các phần tử */
        for(i = 0; i < 3; i++) {
                printf("\n");
                for(j = 0; j < 3; j++)
                        printf("%10p", &A[i][j]);
```

```
    }
    /* Địa chỉ các phần tử truy nhập thông qua con trỏ */
    printf("\n Truy nhập bằng con trỏ:");
    for(i = 0; i < 3; i++) {
        printf("\n");
        for(j = 0; j < 3; j++)
            printf("Địa chỉ phần tử A[%d][%d] = %10p", i, j , (*(A+i)) + j);
    }
    getch();
}
```

Kết quả thực hiện chương trình:

```
Địa chỉ hàng thứ 0 là: FFD2
Địa chỉ hàng thứ 1 là: FFDE
Địa chỉ hàng thứ 2 là: FFEA
Truy nhập bằng con trỏ:
Địa chỉ hàng thứ 0 là: FFD2
Địa chỉ hàng thứ 1 là: FFDE
Địa chỉ hàng thứ 2 là: FFEA
Địa chỉ phần tử A[0][0] = FFD2
Địa chỉ phần tử A[0][1] = FFD6
Địa chỉ phần tử A[0][2] = FFDA
Địa chỉ phần tử A[1][0] = FFDE
Địa chỉ phần tử A[1][1] = FFE2
Địa chỉ phần tử A[1][2] = FFE6
Địa chỉ phần tử A[2][0] = FFEA
Địa chỉ phần tử A[2][1] = FFEE
Địa chỉ phần tử A[2][2] = FFF2
Truy nhập bằng con trỏ:
```

Địa chỉ phần tử A[0][0] = FFD2

Địa chỉ phần tử A[0][1] = FFD6

Địa chỉ phần tử A[0][2] = FFDA

Địa chỉ phần tử A[1][0] = FFDE

Địa chỉ phần tử A[1][1] = FFE2

Địa chỉ phần tử A[1][2] = FFE6

Địa chỉ phần tử A[2][0] = FFEA

Địa chỉ phần tử A[2][1] = FFEE

Địa chỉ phần tử A[2][2] = FFF2

Như vậy, truy nhập mảng nhiều chiều thông qua các phép toán của mảng cũng giống như truy nhập của mảng nhiều chiều thông qua con trỏ. C cung cấp cho người sử dụng một khai báo tương đương với mảng nhiều chiều. Đó là, thay vì khai báo float A[3][3]; chúng ta có thể đưa ra khai báo tương đương float (*A)[3];

Khai báo một con trỏ kiểu float có thể trỏ tới bảng gồm 3 phần tử float. Như vậy, những hàm được truyền vào các mảng nhiều chiều như:

Init_Matrix(float A[N][N], int n);

Có thể được thay thế bằng:

Init_Matrix(float (*A)[N], int n);

Dấu (*A) là cần thiết vì dấu [] có thứ tự ưu tiên cao hơn dấu * và khi đó chương trình dịch sẽ hiểu float *A[N] thành một mảng gồm N con trỏ.

2.7. CON TRỎ TRỎ TỚI CON TRỎ

Như đã được chỉ ra trong ví dụ 2.13. Một mảng được coi như một hằng con trỏ, trong khai báo:

float A[10]; hoàn toàn có thể được thay thế bằng việc sử dụng con trỏ cùng với phép cấp phát bộ nhớ cho con trỏ:

```
float *A;
A = malloc(10 *sizeof(float));
```

Đối với mảng nhiều chiều thì khai báo:

```
float A[3][3];
```

Có thể được thay thế bằng con trỏ tới mảng thông qua khai báo:

```
float (*A)[3];
```

Nhưng chúng ta lại nhận thấy:

```
A[i][j] = (*(A + i))[j] = *((*(A + i)) + j);
```

Hay nói cách khác, truy nhập tới phần tử A[i][j] được thông qua một con trỏ trỏ tới một con trỏ. Điều này có nghĩa ta có thể thay thế một mảng nhiều chiều bằng một con trỏ trỏ tới con trỏ. Nói cách khác, ta có thể khử các hằng trong khai báo của mảng nhiều chiều thông qua một con trỏ trỏ tới một con trỏ.

Ví dụ sau sẽ minh họa cho việc thay thế mảng nhiều chiều bằng con trỏ trỏ tới con trỏ:

Ví dụ 2.14. Viết chương trình tìm định thức của ma trận vuông bằng việc sử dụng con trỏ trỏ tới con trỏ.

```
#include <stdio.h>
#include <stdlib.h>
#include <io.h>
#include <conio.h>
#include <dos.h>
#include <string.h>
#include <alloc.h>
/* Khai báo nguyên mẫu cho hàm */
void Init_Matrix(float **A, int n);
void In_Matrix(float **A, int n);
```

```
int Chuyen(float **A, int n, int i);
float Det_Matrix(float **A, int n);
/* Mô tả hàm */
void Init_Matrix(float **A, int n) {
        int i, j; float temp;
        for(i = 0; i < n; i++) {
                for(j = 0; j < n; j++) {
                        printf("\n Nhập A[%d][%d] = ", i, j);
                        scanf("%f", &temp);
                        A[i][j] = temp;
                }
        }
        In_Matrix(A, n);
}
void In_Matrix(float **A, int n) {
        int i, j;
        for(i = 0; i < n; i++) {
                printf("\n%-3c", 179);
                for(j = 0; j < n; j++) {
                        printf("%8.2f", A[i][j]);
                }
                printf("%3c", 179);
        }
        delay(2000);
}
int chuyen(float **A, int n, int i) {
        float j, k, temp;
        for(j = i+1; j < n; j++) {
```

```
                if(A[j][i]) {
                        for(k = 0; k < n; k++) {
                                temp = A[i][k]; A[i][k] = A[j][k];
                                A[j][k] = temp;
                        }
                        return(1);
                }
        }
        return(0);
}
float Det_Matrix(float **A, int n) {
        float D = 1, p, x; int i, j, k;
        for(i = 0; i < n; i++) {
                if(A[i][i] = = 0) {
                        if(chuyen(A, n, i) = = 0) {
                                printf("\n Định thức D = 0");
                                delay(2000);
                                return(0);
                        }
                        else
                                D = -D;
                }
                for(j = i+1; j < n; j++) {
                        x = A[j][i]/A[i][i];
                        for(k = 0; k < n; k++) {
                                p = A[i][k] *x;
                                A[j][k] = A[j][k]-p;
                        }
```

```
                    In_Matrix(A, n); clrscr();
            }
        }
        for(i = 0; i < n; i++)
            D *= A[i][i];
        printf("\n D = %6.2f", D);
        getch(); return(D);
}
void main(void) {
        float **A; int n, i; clrscr();
        printf("\n Nhập cấp n = "); scanf("%d", &n);
        (*A) = (float *) malloc(n *sizeof(float));
        for(i = 0; i < n; i++)
            A[i] = (float *) malloc(n *sizeof(float));
        Init_Matrix(A, n);
        Det_Matrix(A, n);
        free(A);
}
```

2.8. ĐỐI CỦA HÀM MAIN()

Trong ngôn ngữ C, có một cách để truyền các đối hoặc tham biến như dòng lệnh cho hàm main() ngay từ khi chương trình bắt đầu thực hiện. Khi hàm main() được kích hoạt, nó sẽ gọi tới hai đối. Đối thứ nhất là một biến nguyên được qui định là argc, argc là một biến nguyên có giá trị là số các đối trên dòng lệnh mà chương trình sẽ tạo ra. Đối thứ hai được qui định là argv, argv là con trỏ tới mảng các xâu kí tự, mỗi xâu kí tự trong mảng các xâu kí tự là một đối. Việc thao tác các xâu kí tự này phụ thuộc vào mục tiêu của người sử dụng. Đối thứ nhất của argv (argv[0]) được qui định là tên của chương trình. Ví dụ trong câu lệnh copy của DOS:

copy vanban1.txt vanban2.txt

Thì copy là tên của chương trình cần thực hiện, đối thứ nhất của hàm main là argc có giá trị là 3. Trong đó, argv[0] = "copy", argv[1] = "vanban1.txt", argv[2] = "vanban2.txt". Các đối tùy chọn được phép viết theo bất kỳ trật tự nào, phần còn lại của chương trình sẽ tự động cảm nhận giá trị của argc với số các đối thực tế có mặt.

Chương trình sau là một minh họa đơn giản nhất về các khai báo cần thiết và cách sử dụng đối của hàm main(). Chương trình này chỉ cho hiện lại các đối dòng lệnh của nó trên một dòng, mỗi đối cách nhau bởi các dấu trống. Tức là nếu ta có dòng lệnh:

echo hello, world

Thì chương trình sẽ cho ra:

hello, world

Theo qui ước, argv[0] là tên chương trình, vậy giá trị của argc ít nhất là 1. Trong ví dụ trên, argc có giá trị là 3 và argv[0], argv[1] và argv[2] tương ứng là "echo", "hello" và "world". Đối thực sự đầu tiên là argv[1] và cuối cùng là argv[argc- 1]. Nếu argc là 1 thì sẽ không có đối dòng lệnh nào theo sau tên chương trình. Điều này được trình bày trong chương trình echo.c:

Ví dụ 2.15. In ra các đối của hàm main()

```
#include <stdio.h>
void main(int argc, char *argv[]) {                    /* Hiện các đối của main */
        int i;
        for(i = 1; i < argc; i ++)
                printf("%10s, argv[i]\ n');
}
```

Ví dụ 2.16. Viết chương trình tính tổng của n số thực với đối số được truyền trực tiếp cho hàm main().

```
#include <stdio.h>
float main(int argc, char *argv[]) {
        float     s = 0, k; int i;
        for(i = 1; i < argc; i++) {
                k = atof(argv[i]);
                s += k;
        }
        printf("Tổng s = %6.2f", s);
        return(s);
}
```

Kết quả thực hiện chương trình:

```
TONG 1 2 3 4 5 6 7 8 9
Tổng s = 45.00
```

Ví dụ 2.17. **Viết chương trình sắp xếp tất cả các đối của hàm main() theo thứ tự tăng dần.**

```
#include <stdio.h>
#include <string.h>
void main(int argc, char *argv[]) {
        int i, j; char *temp;
        for(i = 1; i < argc-1; i++) {
                for(j = i+1; j < argc; j++) {
                        if(strcmp(argv[i], argv[j]) > 0) {
                                temp = argv[i]; argv[i] = argv[j];
                                argv[j] = temp;
                        }
                }
```

```
    }
    for(i = 1; i < argc; i++)
        printf("%-10s", argv[i]);
}
```

Kết quả thực hiện chương trình:

```
SAPXEP 98 7 6 5 4 3 2 1 0
1 2 3 4 5 6 7 8 9
```

2.9. CON TRỎ HÀM

Trong ngôn ngữ C, bản thân hàm không phải là một biến, nhưng có thể định nghĩa một con trỏ tới hàm. Thông qua con trỏ hàm, chúng ta có thể coi địa chỉ hàm như một biến. Bằng cách đó hàm có thể được truyền cho hàm hoặc được đặt vào bảng. Điều quan trọng là chúng ta có thể sử dụng con trỏ hàm để quản lý và thực hiện mọi hàm có cùng kiểu với nó. Cũng nhờ con trỏ hàm, chương trình được thể hiện mềm dẻo hơn. Để hiểu rõ bản chất của con trỏ hàm, chúng ta cần phải biết rõ về cơ chế dịch chương trình và lời gọi hàm trong C. Khi dịch chương trình, chương trình dịch trong C chuyển các hàm từ chương trình nguồn thành dạng mã OBJ, mỗi hàm được coi như một mô-đun được cấp phát bộ nhớ tại một vị trí nào đó gọi là địa chỉ của hàm, địa chỉ của hàm là điểm đầu thực hiện của hàm mà tại đó hàm nhận được đầy đủ những tham biến cần thiết để thực hiện.

Khi có một lời gọi hàm trong khi chương trình thực hiện, ngôn ngữ lập trình chuyển điều khiển chương trình tới địa chỉ của hàm tương ứng và thực hiện lời gọi hàm từ chính địa chỉ này. Chương trình dịch của C qui định, tên của hàm đồng thời là địa chỉ của hàm trong bộ nhớ. Do vậy, ta hoàn toàn có thể thực hiện được phép gán con trỏ hàm bằng giá trị là tên của một hàm. Việc khai báo một con trỏ hàm được thực hiện thông qua dòng khai báo sau:

Kiểu_dữ_liệu (*Tên_Hàm)();

Ví dụ:

int (*cmp)(); khai báo con trỏ hàm kiểu int;

float (*fcmp)(); khai báo con trỏ hàm kiểu float;

Ví dụ 2.18. Xây dựng con trỏ hàm đơn giản tính tổng, hiệu, tích, modul của hai số nguyên dương a, b.

```
#include <stdio.h>
int tong(int a, int b) { return(a+b); }
int hieu(int a, int b) { return(a-b); }
int tich(int a, int b) { return(a*b); }
int modul(int a, int b) { return(a%b); }
void main(void) {
        int (*cmp)();                    /* khai báo con trỏ hàm kiểu int */
        int a, b;
        printf("\n Nhập a = "); scanf("%d", &a);
        printf("\n Nhập b = "); scanf("%d", &b);
        cmp = tong;
        printf("\n Tổng a + b = %d", cmp(a, b));
        cmp = hieu;
        printf("\n Hiệu a - b = %d", cmp(a, b));
        cmp = tich;
        printf("\n Tích a * b = %d", cmp(a, b));
        cmp = modul;
        printf("\n modul(a, b) = %d", cmp(a, b));
        getch();
}
```

Nhận xét: Trong ví dụ trên, con trỏ hàm cmp lần lượt nhận địa chỉ của các hàm tong, hieu, tich, modul và thực hiện các hàm này giống

như các nguyên mẫu của nó. Tổng quát, tất cả các hàm có cùng kiểu với cmp đều có thể được quản lý bằng một con trỏ hàm.

Trong ví dụ sau, để tìm bội số chung nhỏ nhất của hai số nguyên dương a và b, chúng ta sẽ sử dụng con trỏ hàm như một biến để truyền cho hàm. Vì bội số chung nhỏ nhất của a và b là thương số của tích a và b cho ước số chung nhỏ nhất của a và b. Do đó, ta hoàn toàn có thể thiết kế được một hàm bao gồm các đối số là a, b và một con trỏ hàm là ước số chung nhỏ nhất của a, b. Chương trình được thể hiện như sau:

Ví dụ 2.19. Tìm bội số chung nhỏ nhất của hai số nguyên dương a, b.

```
#include <stdio.h>
int USCLN(int a, int b) {
        while(a!= b) {
                if(a > b) a = a-b;
                else b = b-a;
        }
        return(a);
}
int BSCNN(int a, int b, int (*cmp)()) {
        int k = cmp(a, b);
        return((a *b)/k);
}
void main(void) {
        int a, b, (*p)();
        printf("\n Nhập a = "); scanf("%d", &a);
        printf("\n Nhập b = "); scanf("%d", &b);
        p = USCLN;
        printf("\n BSCNN = %d", BSCNN(a, b, p));
        getch();
}
```

BÀI TẬP CHƯƠNG 2

2.1. Viết chương trình tính giá trị đa thức P(x) tại điểm x_0.

2.2. Viết chương trình tính đạo hàm cấp n của đa thức P(x) bậc n.

2.3. Viết chương trình tính tổng hai đa thức P(x) bậc n và Q(x) bậc m.

2.4. Viết chương trình tính hiệu hai đa thức P(x) bậc n và Q(x) bậc m.

2.5. Viết chương trình tính tích hai đa thức P(x) bậc n và Q(x) bậc m.

2.6. Viết chương trình tính đa thức thương và đa thức phần dư của đa thức P(x) bậc n và Q(x) bậc m.

2.7. **Thuật toán cộng.** Sử dụng thuật toán cộng xây dựng chương trình cộng hai số nguyên a và b có biểu diễn nhị phân tương ứng là $a = (a_1, a_2..., a_n)$, $b = (b_1, b_2, ..., b_n)$, $0 < n \leq 80$. Dữ liệu vào cho bởi tệp cong.inp, dòng đầu tiên ghi lại số tự nhiên a, dòng kế tiếp ghi lại số tự nhiên b. Kết quả ghi lại trong tệp cong.out. Ví dụ sau sẽ minh họa cho tệp cong.in và cong.out.

cong.in	cong.out
1234	6912
5678	

2.8. **Thuật toán nhân.** Sử dụng thuật toán cộng trong bài 2.7 xây dựng chương trình nhân hai số nguyên a và b có biểu diễn nhị phân tương ứng là $a = (a_1, a_2. ..., a_n)$, $b = (b_1, b_2, ..., b_n)$, $0 < n \leq 80$. Dữ liệu vào cho bởi tệp nhan.inp, kết quả ghi lại trong tệp nhan.out.

2.9. Xây dựng chương trình mô tả lại các phép toán cộng, nhân bằng tay. Dữ liệu vào cho bởi tệp pheptoan.inp, kết quả ghi lại trong tệp pheptoan.out.

2.10. Xây dựng tập các thao tác cộng, trừ, nhân, chia nguyên, lấy phần dư cho các số lớn. Dữ liệu vào cho bởi tệp solon.inp, kết quả ghi lại trong tệp solon.out.

2.11. Sử dụng thuật toán xác suất tìm số nguyên tố để tìm các số nguyên tố cực lớn có độ dài ít nhất là 512 số. (Bài tập không bắt buộc).

2.12. **Bao đóng.** Cho tập hợp $U \subseteq [A. .Z]$ được gọi là tập thuộc tính, $X \subseteq U$. F là tập phụ thuộc hàm trên U được định nghĩa như sau:

$F = \{Z \text{ - > } W$ với $Z \subseteq U$, $W \subseteq U\}$. Kí hiệu -> được hiểu là dẫn được.

Ta gọi X^+ là bao đóng của tập thuộc tính X trên tập phụ thuộc hàm F được xây dựng theo thuật toán đệ qui sau:

i. $X^{(0)} = X$

ii. $X^i = X^{i-1} \cup W$ với $Z \subseteq U$, $W \subseteq U$ và $Z \text{ -> } W \in F$

iii. Nếu $X^k = X^{k-1}$ thì $X^+ = X^k$ với mọi phép thử trên mỗi phụ thuộc hàm (mỗi phụ thuộc hàm được thử ít nhất một lần mà không bổ sung thêm được thuộc tính mới).

Cho tệp dữ liệu baodong.in được tổ chức như sau:

Dòng đầu tiên ghi lại một số tự nhiên N là số các phụ thuộc hàm trong tập phụ thuộc hàm F, dòng kế tiếp là một xâu kí tự ghi lại tập thuộc tính U, N dòng tiếp theo ghi các phụ thuộc hàm, mỗi phụ thuộc hàm được ghi trên một dòng, dòng cuối cùng là tập thuộc tính $X \subseteq U$.

Hãy tìm bao đóng của tập phụ thuộc hàm X và ghi lại kết quả vào tệp baodong.out.

Ví dụ:

tệp baodong.in	baodong.out
5	
ABCDEFGHIJ	ABCDEFGHIJ
A - > BCD	
BC - > CD	
E - > FGH	
EF - > FGH	
AE - > IJ	
AE	

2.13. Khoá. Cho lược đồ quan hệ $\propto = < U, F >$ trong đó U là tập thuộc tính, F là tập phụ thuộc hàm, $K \subseteq U$. K được gọi là khoá của $\propto = < U, F >$ nếu:

Bao đóng của K bằng U (K+= U);

Với mọi thuộc tính $A \in K$ thì $(K \backslash A)^+ \neq U$.

Hãy tìm một khoá của lược đồ quan hệ $\propto = < U, F >$. Dữ liệu vào cho bởi tệp khoa.in, dòng đầu tiên ghi lại số tự nhiên n là số các phụ thuộc hàm, dòng kế tiếp ghi lại tập thuộc tính U, n dòng tiếp theo ghi lại các phụ thuộc hàm, mỗi phụ thuộc hàm được viết trên một dòng, vế trái và vế phải của mỗi phụ thuộc hàm được phân biệt với nhau bởi một hoặc vài ký tự rỗng. Kết quả ghi lại trong tệp khoa.out. Ví dụ sau sẽ minh họa cho tệp khoa.in và khoa.out.

khoa.in	khoa.out
3	AE
ABCDEFGHIJ	

A BCD

E FGH

AE IJ

2.14. **Phủ.** Cho lược đồ quan hệ $\propto = < U, F >$ và $\propto = < U, G >$. F được gọi là phủ của G nếu với mọi f: X -> Y $\in$ F thì $X^+_G \supseteq Y$ và với mọi g: X -> Y $\in$ G thì X + F $\supseteq$ Y. Hãy kiểm tra tính tương đương của hai tập phụ thuộc hàm F/U và G/U. Dữ liệu vào cho bởi tệp phu.in, kết quả ghi lại trong tệp phu.out.

2.15. **Phủ tự nhiên.** Cho lược đồ quan hệ $\propto = <U, F>$, và tập phụ thuộc hàm G/U. G được gọi là phủ tự nhiên của F nếu:

a) G là phủ của F;

b) Với mọi f: $X \rightarrow Y \in G$ thì $X \cap Y = \Phi$;

c) Với mọi $X \rightarrow Y, Z \rightarrow W \in F$ thì $X \neq Z$.

Hãy tìm một phủ tự nhiên của F. Dữ liệu vào cho bởi tệp phutn.in, kết quả ghi lại trong tệp phutn.out.

2.16. **Phủ không dư.** Cho lược đồ quan hệ $\propto = < U, F >$, và tập phụ thuộc hàm G/U. G được gọi là phủ không dư của F nếu:

a) G là phủ của F;

b) Với mọi g: $X \rightarrow Y \in G$ thì $\{G \setminus g\}$ không là phủ của G.

Xuất phát từ một phủ tự nhiên, hãy tìm một phủ không dư của F. Dữ liệu vào cho bởi tệp phukd.in, kết quả ghi lại trong tệp phukd.out.

2.17. Cho tệp dữ liệu dathuc.in được tổ chức như sau:

- Dòng đầu tiên ghi lại bốn số nguyên, n, m, x_0, l, mỗi số cách nhau bởi một và dấu trống;
- Dòng kế tiếp ghi lại n +1 số thực là hệ số của đa thức P(x) bậc n;

- Dòng thứ 3 ghi lại m + 1 số thực là hệ số của đa thức Q(x) bậc m.

Hãy viết chương trình đưa kết quả ra màn hình và ghi vào tệp dathuc.out với những thông tin sau:

- Tính giá trị của các đa thức $P(x_0)$, $Q(x_0)$ ghi ở dòng thứ nhất mỗi số cách nhau một hay và dấu trống;
- Tính đạo hàm cấp l < n, l < m của đa thức P(x), Q(x) và ghi vào trong Tệp kết quả ở hai dòng kế tiếp;
- Dòng tiếp theo là đa thức tổng R(x) = P(x) + Q(x);
- Dòng tiếp theo là đa thức hiệu R(x) = P(x) - Q(x);
- Dòng tiếp theo là đa thức tích R(x) = P(x) / Q(x);
- Hai dòng cuối cùng là đa thức thương và đa thức phần dư của phép chia P(x)/Q(x).

Ví dụ về tệp dathuc.in:

```
3   2   1   1
1   3   3   1
1   2   1
```

và dathuc.out

```
8   4
3   6   3
2   2
1   4   5   2
1   2   1   0
1   5   10  10  5   1
1   1
0   0
```

2.18. Cho ma trận A và cấp m × n. Hãy tìm tổng, hiệu của hai ma trận A và B.

2.19. Cho ma trận A cấp m × n và ma trận B cấp n × m. Hãy tính tích của hai ma trận A và B.

2.20. Cho tệp dữ liệu matrix.in được tổ chức theo khuôn dạng như sau:

- Dòng đầu tiên là một số tự nhiên n là cấp của ma trận vuông A;
- n dòng tiếp theo mỗi dòng ghi n số thực. Mỗi số thực được phân biệt với nhau bởi một hoặc vài kí tự trống là các phần tử A[i][j] của ma trận vuông A;

Hãy viết chương trình tìm hàng, cột hoặc đường chéo có tổng các phần tử là lớn nhất. Ghi kết quả hàng, cột hoặc đường chéo vào tệp matrix.out mỗi phần tử được phân biệt bởi một vài ký tự trống.

Ví dụ tệp matrix.in

```
3
1    2    4
4    8    12
3    -3   0
```

Tệp matricx.out

```
4    8    12
```

2.21. Cho tệp dữ liệu matrix.in được tổ chức theo khuôn dạng như sau:

- Dòng đầu tiên là một số tự nhiên n là cấp của ma trận vuông A;
- n dòng tiếp theo mỗi dòng ghi n số thực. Mỗi số thực được phân biệt với nhau bởi một hoặc vài kí tự trống là các phần tử A[i][j] của ma trận vuông A;

Tìm định thức của A và ghi lại kết quả vào tệp matrix.out.

Ví dụ tệp matrix.in

```
3
1   2   4
4   8   12
3   -3  0
```

và tệp matrix.out

```
-36
```

2.22. Cho tệp dữ liệu matrix.in được tổ chức theo khuôn dạng như sau:

- Dòng đầu tiên là một số tự nhiên n là cấp của ma trận vuông A;
- n dòng tiếp theo mỗi dòng ghi n số thực mỗi số thực được phân biệt với nhau bởi một hoặc vài kí tự trống là các phần tử A[i][j] của ma trận vuông A;

Hãy viết chương trình tìm ma trận nghịch đảo của ma trận vuông A. Ghi lại kết quả vào tệp matrix.in theo từng dòng. Trong trường hợp ma trận không tồn tại nghịch đảo ghi lại thông báo "Nghịch đảo không tồn tại".

Ví dụ tệp matrix.in

```
3
1   2   3
2   5   3
1   0   8
```

Và tệp matrix.out

```
-40   16   9
```

13	-5	-3
5	-2	-1

2.23. Cho tệp dữ liệu matrix.in được tổ chức theo khuôn dạng như sau:

- Dòng đầu tiên là một số tự nhiên n là cấp của ma trận vuông A;
- n dòng tiếp theo mỗi dòng ghi n + 1 số thực mỗi số thực được phân biệt với nhau bởi một hoặc vài kí tự trống là các phần tử A[i][j] của ma trận vuông A và hệ số của véc tơ n thành phần B;

Hãy viết chương trình giải hệ phương trình tuyến tính thuần nhất n ẩn AX = B bằng phương pháp khử Gaus. Nghiệm của hệ được ghi lại trong tệp matrix.out trên một dòng. Trong trường hợp hệ suy biến. Ghi lại thông báo "Hệ suy biến".

Ví dụ về tệp matrix.in

3			
1	1	-1	1
2	-1	1	0
3	1	-2	1

và tệp matrix.out

0.33	1.33	0.66

2.24. Ma trận nhị phân là ma trận mà các phần tử của nó hoặc bằng 0 hoặc bằng 1. Đối với ma trận nhị phân người ta định nghĩa các phép hợp, giao, nhân logic và luỹ thừa như sau:

- **Phép hợp.** Cho hai ma trận nhị phân cùng cấp m × n, A = { a_{ij} }, B = { b_{ij} } (i = 1, 2. ., m, j = 1, 2, . ., n). Hợp của A với B cho ta ma trận C = { c_{ij} | $c_{ij} = a_{ij} \vee b_{ij}$; i = 1, 2, . ., m; j = 1, 2, . ., n}.

Ví dụ:

$$A=\begin{vmatrix}1 & 0 & 1\\ 0 & 1 & 0\end{vmatrix} \quad B=\begin{vmatrix}0 & 1 & 0\\ 1 & 1 & 0\end{vmatrix}$$

$$C=A\vee B=\begin{vmatrix}1 & 1 & 1\\ 1 & 1 & 0\end{vmatrix}$$

- **Phép giao.** Phép giao của hai ma trận $A = \{a_{ij}\}$ và $B = \{b_{ij}\}$ cùng cấp $m \times n$ ho ta một ma trận cùng cấp $C = \{c_{ij} \mid c_{ij} = a_{ij} \wedge b_{ij}\}$; $i = 1, 2, ..., m$; $j = 1, 2, ..., n$.

Ví dụ:

$$A=\begin{vmatrix}1 & 0 & 1\\ 0 & 1 & 0\end{vmatrix} \quad B=\begin{vmatrix}0 & 1 & 0\\ 1 & 1 & 0\end{vmatrix}$$

$$C=A\vee B=\begin{vmatrix}0 & 0 & 0\\ 0 & 1 & 0\end{vmatrix}$$

- **Phép nhân logic.** Cho ma trận nhị phân $A = \{a_{ij}\}$ cấp $m \times k$ và $B = \{b_{ij}\}$ cấp $k \times n$. Ma trận $C = \{ c_{ij} \mid c_{ij} = (a_{i1} \wedge b_{1j}) \vee (a_{i2} \wedge b_{2j}) \vee ... \vee (a_{ik} \wedge b_{kj})\}$ cấp $m \times n$ được gọi là ma trận tích logic của ma trận A với ma trận B và được ký hiệu là $C = A\Theta B$.

Ví dụ:

$$A=\begin{vmatrix}1 & 0\\ 0 & 1\\ 1 & 0\end{vmatrix} \quad B=\begin{vmatrix}1 & 1 & 0\\ 0 & 1 & 1\end{vmatrix}$$

$$C = A\,\Theta\,B = \begin{vmatrix}1 & 1 & 0\\ 0 & 1 & 1\\ 1 & 1 & 0\end{vmatrix}$$

- **Phép luỹ thừa logic bậc r.** Luỹ thừa logic bậc r của ma trận vuông nhị phân A, ký hiệu A[r] = A Θ A ΘA Θ ...ΘA (r lần).

Bài toán: Cho hai ma trận vuông nhị phân cấp n A = $\{a_{ij}\}$ và B = $\{b_{ij}\}$. Hãy viết chương trình tính hợp, giao, nhân logic, và luỹ thừa bậc r của A và B.

2.25. Cho một bảng hình vuông gồm 5 x 5 hình vuông nhỏ. Trên mỗi ô ghi một chữ số thập phân như hình vẽ sau:

3	5	1	1	1
5	0	0	3	3
1	0	3	4	3
1	3	4	2	1
1	3	3	1	3

Các chữ số điền vào phải thoả mãn những yêu cầu sau:

Yêu cầu 1: 12 số bao gồm năm số năm chữ số theo từng dòng đọc từ trái sang phải, năm số năm chữ số theo từng cột đọc từ trên xuống dưới, hai số năm chữ số theo hai đường chéo đọc từ trái sang phải đều là các số nguyên tố với 5 chữ số có nghĩa. 12 số này không nhất thiết phải khác nhau.

Yêu cầu 2: Tổng các chữ số của mỗi số trong 12 số trên đều bằng nhau. Trong ví dụ trên, tổng các chữ số của mỗi số đều bằng 11.

Hãy viết chương trình thực hiện các công việc sau:

Nhập dữ liệu từ tệp ngto.in. Trong đó, dòng thứ nhất ghi số là tổng các chữ số của những số cần xây dựng, dòng thứ hai ghi chữ số ở góc bên trái hình vuông.

Tìm các lời giải có thể có ứng với dữ liệu vào đã nhập vào tệp ngto.out. Một lời giải là một mảng 5 x 5 ghi thành 5 dòng mỗi dòng 5 chữ số tương ứng với các hàng của hình vuông, hai lời giải được ghi cách nhau bởi một dòng trống, hai lời giải khác

nhau nếu hai hình vuông có ít nhất một ô khác nhau. Trong ví dụ trên, ta có các tệp input và output như sau:

ngto.in	ngto.out				
11	3	5	1	1	1
3	5	0	0	3	3
	1	0	3	4	3
	1	3	4	2	1

CHƯƠNG 3

DUYỆT VÀ ĐỆ QUI

3.1. ĐỊNH NGHĨA BẰNG ĐỆ QUI

Trong thực tế, chúng ta gặp rất nhiều đối tượng mà khó có thể định nghĩa nó một cách tường minh, nhưng lại dễ dàng định nghĩa đối tượng qua chính nó. Kỹ thuật định nghĩa đối tượng qua chính nó được gọi là kỹ thuật đệ qui (recursion). Đệ qui được sử dụng rộng rãi trong khoa học máy tính và lý thuyết tính toán. Các giải thuật đệ qui đều được xây dựng thông qua hai bước: bước phân tích và bước thay thế ngược lại.

Ví dụ 3.1. Để tính tổng S(n) = 1 + 2 + ... + n, chúng ta có thể thực hiện thông qua hai bước như sau:

Bước phân tích:

Để tính toán được S(n) trước tiên ta phải tính toán trước S(n - 1), sau đó tính S(n) = S(n - 1) + n.

- Để tính toán được S(n - 1), ta phải tính toán trước S(n - 2) sau đó tính S(n - 1) = S(n - 2) + n - 1.
- Tiếp tục tính S(n - 2), S(n - 3) ...
- Để tính toán được S(2), ta phải tính toán trước S(1) sau đó tính S(2) = S(1) + 2.
- Và cuối cùng S(1) chúng ta có ngay kết quả là 1

Bước thay thế ngược lại:

Xuất phát từ S(1), thay thế ngược lại chúng ta xác định S(n):

- S(1) = 1
- S(2) = S(1) + 2
- S(3) = S(2) + 3

- .....
- S(n) = S(n - 1) + n

Ví dụ 3.2. Định nghĩa hàm bằng đệ qui

Hàm f(n) = n!

Dễ thấy f(0) = 1.

Vì (n + 1)! = 1 . 2.3 ... n.(n + 1) = n!(n + 1), nên ta có:

f(n + 1) = (n + 1).f(n) với mọi n nguyên dương.

Hàm f(n) = a^n

Vì $a^0 = 1$; $f(n + 1) = a^{n+1} = a.a^n = a.f(n)$ nên f(n) = a.f(n) với mọi số thực a và số tự nhiên n.

Ví dụ 3.3. Tập hợp định nghĩa bằng đệ qui

Định nghĩa đệ qui tập các xâu: Giả sử Σ^* là tập các xâu trên bộ chữ cái Σ. Khi đó Σ^* được định nghĩa bằng đệ qui như sau:

$\lambda \in \Sigma^*$, trong đó λ là xâu rỗng

$wx \in \Sigma^*$ nếu $w \in \Sigma^*$ và $x \in \Sigma^*$

Ví dụ 3.4. Cấu trúc tự trỏ được định nghĩa bằng đệ qui

```
struct node {
    int infor;
    struct node *left;
    struct node *right;
};
```

3.2. GIẢI THUẬT ĐỆ QUI

Một thuật toán được gọi là đệ qui nếu nó giải bài toán bằng cách rút gọn bài toán ban đầu thành bài toán tương tự như vậy sau một số hữu hạn lần thực hiện. Trong mỗi lần thực hiện, dữ liệu đầu vào tiệm cận tới tập dữ liệu dừng.

Ví dụ: Để giải quyết bài toán tìm ước số chung lớn nhất của hai số nguyên dương a và b với b > a, ta có thể rút gọn về bài toán tìm ước số chung lớn nhất của (b mod a) và a vì USCLN(b mod a, a) = USCLN(a, b). Dãy các rút gọn liên tiếp có thể đạt được cho tới khi đạt điều kiện dừng USCLN(0, a) = USCLN(a, b) = a. Sau đây là ví dụ về một số thuật toán đệ qui thông dụng.

Thuật toán 1: Tính a^n bằng giải thuật đệ qui, với mọi số thực a và số tự nhiên n.

```
double power(float a, int n) {
    if(n == 0)
        return(1);
    return(a *power(a, n - 1));
}
```

Thuật toán 2: Thuật toán đệ qui tính ước số chung lớn nhất của hai số nguyên dương a và b.

```
int USCLN(int a, int b) {
    if(a == 0)
        return(b);
    return(USCLN(b % a, a));
}
```

Thuật toán 3: Thuật toán đệ qui tính n!

```
long factorial(int n) {
    if(n == 1)
        return(1);
    return(n * factorial(n - 1));
}
```

Thuật toán 4: Thuật toán đệ qui tìm kiếm nhị phân:

```
int binary_search(float *A, int x, int i, int j) {
    int mid = (i + j)/2;
```

```
    if(x > A[mid] && i < mid)
        return(binary_search(A, x, mid + 1, j);
    else if(x < A[mid] && j > mid)
        return(binary_search(A, x, i, mid - 1);
    else if(x == A[mid])
        return(mid);
    return(-1);
}
```

Thuật toán 5: Thuật toán đệ qui tính số fibonacci thứ n

```
int fibonacci(int n) {
  if(n == 0) return(0);
      else if(n == 1) return(1);
      return(fibonacci(n - 1) + fibonacci(n - 2));
}
```

3.3. THUẬT TOÁN SINH KẾ TIẾP

Phương pháp sinh kế tiếp dùng để giải quyết bài toán liệt kê của lý thuyết tổ hợp. Thuật toán sinh kế tiếp chỉ được thực hiện trên lớp các bài toán thỏa mãn hai điều kiện sau:

- Có thể xác định được một thứ tự trên tập các cấu hình tổ hợp cần liệt kê, từ đó xác định được cấu hình đầu tiên và cấu hình cuối cùng.
- Từ một cấu hình bất kỳ chưa phải là cuối cùng, đều có thể xây dựng được một thuật toán để suy ra cấu hình kế tiếp.

Tổng quát, thuật toán sinh kế tiếp có thể được mô tả bằng thủ tục generate, trong đó Sinh_Ke_Tiep là thủ tục sinh cấu hình kế tiếp theo thuật toán sinh đã được xây dựng. Nếu cấu hình hiện tại là cấu hình cuối cùng thì thủ tục Sinh_Ke_Tiep sẽ gán cho *stop* giá trị TRUE, ngược lại cấu hình kế tiếp sẽ được sinh ra.

```
Procedure Generate;
  begin
     < Xây dựng cấu hình ban đầu >
     stop : = false;
     while not stop do
     begin
        < Đưa ra cấu hình đang có >;
        Sinh_Ke_Tiep;
     end;
  end;
```

Sau đây chúng ta xét một số ví dụ minh họa cho thuật toán sinh.

3.3.1. Bài toán liệt kê các tập con của tập n phần tử

Một tập hợp hữu hạn gồm n phần tử đều có thể biểu diễn tương đương với tập các số tự nhiên 1, 2, ..., n. Bài toán được đặt ra là: Cho một tập hợp gồm n phần tử $X = \{X_1, X_2, ..., X_n\}$, hãy liệt kê tất cả các tập con của tập hợp X.

Để liệt kê được tất cả các tập con của X, ta làm tương ứng mỗi tập $Y \subseteq X$ với một xâu nhị phân có độ dài n là $B = \{B_1, B_2, ..., B_n\}$ sao cho $B_i = 0$ nếu $X_i \notin Y$ và $B_i = 1$ nếu $X_i \in Y$; như vậy, phép liệt kê tất cả các tập con của một tập hợp n phần tử tương đương với phép liệt kê tất cả các xâu nhị phân có độ dài n. Số các xâu nhị phân có độ dài n là 2^n. Bây giờ ta đi xác định thứ tự các xâu nhị phân và phương pháp sinh kế tiếp.

Nếu xem các xâu nhị phân $b = \{b_1, b_2, ..., b_n\}$ như là biểu diễn của một số nguyên dương p(b). Khi đó thứ tự hiển nhiên nhất là thứ tự tự nhiên được xác định như sau:

Ta nói xâu nhị phân $b = \{b_1, b_2, ..., b_n\}$ có thứ tự trước xâu nhị phân $b' = \{b'_1, b'_2, ..., b'_n\}$ và kí hiệu là $b < b'$ nếu $p(b) < p(b')$. Ví dụ với n = 4: chúng ta có $2^4 = 16$ xâu nhị phân (tương ứng với 16 tập con của tập gồm n phần tử) được liệt kê theo thứ tự từ điển như sau:

b	p(b)
0 0 0 0	0
0 0 0 1	1
0 0 1 0	2
0 0 1 1	3
0 1 0 0	4
0 1 0 1	5
0 1 1 0	6
0 1 1 1	7
1 0 0 0	8
1 0 0 1	9
1 0 1 0	10
1 0 1 1	11
1 1 0 0	12
1 1 0 1	13
1 1 1 0	14
1 1 1 1	15

Từ đây ta xác định được xâu nhị phân đầu tiên là 00...00 và xâu nhị phân cuối cùng là 11...11. Quá trình liệt kê dừng khi ta được xâu nhị phân 1111. Xâu nhị phân kế tiếp là biểu diễn nhị phân của giá trị xâu nhị phân trước đó cộng thêm 1 đơn vị. Từ đó ta nhận được qui tắc sinh kế tiếp như sau:

- Tìm chỉ số i đầu tiên theo thứ tự i = n, n - 1, ..., 1 sao cho $b_i = 0$.
- Gán lại $b_i = 1$ và $b_j = 0$ với tất cả j > i. Dãy nhị phân thu được là dãy cần tìm

Thuật toán sinh xâu nhị phân kế tiếp

```
void Next_Bit_String(int *B, int n) {
    i = n;
```

```
            while(b_i == 1) {
                b_i = 0
                i = i - 1;
            }
            b_i = 1;
        }
```

Sau đây là văn bản chương trình liệt kê các xâu nhị phân có độ dài n:

```
#include <stdio.h>
#include <alloc.h>
#include <stdlib.h>
#include <conio.h>
#define MAX 100
#define TRUE 1
#define FALSE 0
int Stop, count;
void Init(int *B, int n) {
    int i;
    for(i = 1; i <= n ; i ++)
        B[i] = 0;
    count = 0;
}
void Result(int *B, int n) {
    int i; count ++;
    printf("\n Xâu nhị phân thứ %d:", count);
    for(i = 1; i <= n; i ++)
            printf("%3d", B[i]);
}
```

```
void Next_Bits_String(int *B, int n) {
    int i = n;
    while(i > 0 && B[i]) {
            B[i] = 0; i -- ;
    }
     if(i == 0)
        Stop = TRUE;
     else
        B[i] = 1;
}
void Generate(int *B, int n) {
    int i;
    Stop = FALSE;
    while(!Stop) {
        Result(B, n);
        Next_Bits_String(B, n);
    }
}
void main(void) {
    int i, *B, n; clrscr();
    printf("\n Nhập n = "); scanf("%d", &n);
    B = (int *) malloc(n*sizeof(int));
    Init(B, n);
    Generate(B, n);
    free(B);
    getch();
}
```

3.3.2. Bài toán liệt kê tập con m phần tử của tập n phần tử

Bài toán: Cho tập hợp X = {1, 2, ..., n}. Hãy liệt kê tất cả các tập con m < n phần tử của X.

Mỗi tập con m phần tử của X có thể biểu diễn như một bộ có thứ tự:

$a = (a_1, a_2, ..., a_m)$ thoả mãn $1 \le a_1 < a_2 < ... < a_m \le n$

Trên các tập con m phần tử của X, ta định nghĩa thứ tự của các tập con như sau:

Ta nói tập con $a = (a_1, a_2, ..., a_m)$ có thứ tự trước tập $a' = (a'_1, a'_2, ..., a'_m)$ theo thứ tự từ điển và kí hiệu $a < a'$ nếu tìm được chỉ số k sao cho:

$a_1 = a'_1, a_2 = a'_2 .., a_{k-1} = a'_{k-1}, a_k < a'_k$

Ví dụ X = {1, 2, 3, 4, 5}, n = 5, m = 3. Các tập con 3 phần tử của X được liệt kê theo thứ tự từ điển như sau:

1 2 3
1 2 4
1 2 5
1 3 4
1 3 5
1 4 5
2 3 4
2 3 5
2 4 3
3 4 5

Như vậy, tập con đầu tiên là 1, 2, ..., m. Tập con cuối cùng là n - m + 1, n - m + 2, ..., n. Giả sử ta có tập con chưa phải là cuối cùng (nhỏ hơn so với tập con n - m + 1, n - m + 2, ..., n), khi đó tập con kế tiếp của a được sinh bởi các qui tắc biến đổi sau:

- Tìm từ bên phải dãy $a_1, a_2, ..., a_m$ phần tử $a_i \ne n - m + i$,

- Thay thế $a_i = a_i + 1$;
- Thay a_j bởi $a_i + j - i$, với $j = i + 1, i + 2, ..., m$.

Với qui tắc sinh như trên, chúng ta có thể mô tả bằng thuật toán sau:

Thuật toán liệt kê tập con kế tiếp m phần tử của tập n phần tử:

```
void Next_Combination(int *A, int m) {
    i = m;
    while(a_i == m - n + i)
        i = i - 1;
    a_i = a_i + 1;
    for(j = i + 1; j <= m; j ++)
        a_j = a_i + j - i;
}
```

Chương trình liệt kê tập các tập con m phần tử của tập n phần tử được thể hiện như sau:

```
#include <stdio.h>
#include <conio.h>
#define TRUE 1
#define FALSE 0
#define MAX 100
int n, k, count, C[MAX], Stop;
void Init(void) {
    int i;
    printf("\n Nhập n = "); scanf("%d", &n);
    printf("\n Nhập k = "); scanf("%d", &k);
    for(i = 1; i <= k; i ++)
        C[i] = i;
}
```

```
void Result(void) {
    int i; count ++;
    printf("\n Tập con thứ %d:", count);
    for(i = 1; i <= k; i ++)
        printf("%3d", C[i]);
}
void Next_Combination(void) {
    int i, j;
    i = k;
    while(i > 0 && C[i] == n - k + i)
        i --;
    if(i > 0) {
        C[i] = C[i] + 1;
        for(j = i + 1; j <= k; j ++)
            C[j] = C[i] + j - i;
    }
    else Stop = TRUE;
}
void Combination(void) {
    Stop = FALSE;
    while(!Stop) {
        Result();
        Next_Combination();
    }
}
void main(void) {
    clrscr();
    Init();
```

```
    Combination();
    getch();
}
```

3.3.3. Bài toán liệt kê các hoán vị của tập n phần tử

Bài toán: Cho tập hợp X = {1, 2, ..., n}. Hãy liệt kê tất cả các hoán vị của X.

Mỗi hoán vị n phần tử của tập hợp X có thể biểu diễn bởi bộ có thứ tự gồm n thành phần $a = (a_1, a_2, .., a_n)$ thoả mãn:

$a_i \in X$; $i = 1, 2, ..., n$; $a_p \neq a_q$ nếu $p \neq q$.

Trên tập có thứ tự các hoán vị n phần tử của X, ta định nghĩa thứ tự của các hoán vị đó như sau:

Hoán vị $a = (a_1, a_2, ..., a_n)$ được gọi là có thứ tự trước hoán vị $a' = (a'_1, a'_2, ..., a'_n)$ và kí hiệu $a < a'$ nếu tìm được chỉ số k sao cho:

$a_1 = a'_1, a_2 = a'_2, ..., a_{k-1} = a'_{k-1}, a_k < a'_k$

Ví dụ X = {1, 2, 3, 4} khi đó các hoán vị của 4 phần tử được sắp xếp theo thứ tự từ điển như sau:

1	2	3	4
1	2	4	3
1	3	2	4
1	3	4	2
1	4	2	3
1	4	3	2
2	1	3	4
2	1	4	3
2	3	1	4
2	3	4	1
2	4	1	3
2	4	3	1

3	1	2	4
3	1	4	2
3	2	1	4
3	2	4	1
3	4	1	2
3	4	2	1
4	1	2	3
4	1	3	2
4	2	1	3
4	2	3	1
4	3	1	2
4	3	2	1

Như vậy, hoán vị đầu tiên là 1, 2, ..., n, hoán vị cuối cùng là n, n - 1, ...1. Giả sử hoán vị a = (a_1, a_2, ..., a_n) chưa phải là hoán vị cuối cùng, khi đó hoán vị kế tiếp của a được sinh bởi qui tắc sau:

- Tìm từ phải qua trái hoán vị có chỉ số j đầu tiên thoả mãn $a_j < a_{j+1}$ hay j là chỉ số lớn nhất để $a_j < a_{j+1}$;
- Tìm a_k là số nhỏ nhất còn lớn hơn a_j trong các số ở bên phải a_j;
- Đổi chỗ a_j cho a_k;
- Lật ngược lại đoạn từ a_{j+1}, ..., a_n.

Thuật toán sinh hoán vị kế tiếp:

```
void Next_Permutation(int *A, int n) {
    int j, k, r, s, temp;
    j = n;
    while(a_j > a_{j+1})
        j = j - 1;
    k = n;
    while(a_j > a_k)
```

```
            k = k - 1;
        temp = a_j; a_j = a_k; a_k = temp;
        r = j + 1; s = n;
        while(r < s) {
            temp = a_r; a_r = a_s; a_s = temp;
            r = r + 1;  s = s - 1;
        }
    }
```

Chương trình liệt kê các hoán vị của tập hợp gồm n phần tử như sau:

```
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#define MAX 20
#define TRUE 1
#define FALSE 0
int P[MAX], n, count, Stop;
void Init(void) {
    int i; count = 0;
    printf("\n Nhập n = "); scanf("%d", &n);
    for(i = 1; i <= n; i ++)
        P[i] = i;
}
void Result(void) {
    int i; count ++;
    printf("\n Hoán vị %d:", count);
    for(i = 1; i <= n; i ++)
        printf("%3d", P[i]);
}
void Next_Permutation(void) {
```

```
    int j, k, r, s, temp;
    j = n - 1;
    while(j > 0 && P[j] > P[j + 1])
        j -- ;
    if(j == 0)
        Stop = TRUE;
    else {
        k = n;
        while(P[j] > P[k]) k -- ;
        temp = P[j]; P[j] = P[k]; P[k] = temp;
        r = j + 1; s = n;
        while(r < s) {
            temp = P[r]; P[r] = P[s]; P[s] = temp;
            r ++; s -- ;
        }
    }
}
void Permutation(void) {
    Stop = FALSE;
    while(!Stop) {
        Result();
        Next_Permutation();
    }
}
void main(void) {
    Init();
    clrscr();
    Permutation();
    getch();
}
```

3.3.4. Bài toán chia số tự nhiên n thành tổng các số nhỏ hơn

Bài toán: Cho n là số nguyên dương. Một cách phân chia số n là biểu diễn n thành tổng các số tự nhiên không lớn hơn n. Chẳng hạn 8 = 2 + 3 + 2.

Hai cách chia được gọi là đồng nhất nếu chúng có cùng các số hạng và chỉ khác nhau về thứ tự sắp xếp. Bài toán được đặt ra là, cho số tự nhiên n, hãy duyệt mọi cách phân chia số n.

Chọn cách phân chia số $n = b_1 + b_2 + ... + b_k$ với $b_1 > b_2 > ... > b_k$, và duyệt theo trình tự từ điển ngược. Chẳng hạn với n = 7, chúng ta có thứ tự từ điển ngược của các cách phân chia như sau:

7						
6	1					
5	2					
5	1	1				
4	3					
4	2	1				
4	1	1	1			
3	3	1				
3	2	2				
3	2	1	1			
3	1	1	1	1		
2	2	2	1			
2	2	1	1	1		
2	1	1	1	1	1	
1	1	1	1	1	1	1

Như vậy, cách chia đầu tiên chính là n. Cách chia cuối cùng là dãy n số 1 Bây giờ chúng ta chỉ cần xây dựng thuật toán sinh kế tiếp cho mỗi cách phân chia chưa phải là cuối cùng.

Thuật toán sinh cách phân chia kế tiếp:

```
void Next_Division(void) {
  int i, j, R, S, D;
  i = k;
  while(i > 0 && C[i] == 1)
      i --;
  if(i > 0) {
      C[i] = C[i] - 1;
      D = k - i + 1;
      R = D / C[i];
      S = D % C[i];
      k = i;
      if(R > 0) {
          for(j = i + 1; j <= i + R; j ++)
              C[j] = C[i];
          k = k + R;
      }
      if(S > 0) {
          k = k + 1; C[k] = S;
      }
  }
  else Stop = TRUE;
}
```

Chương trình được thể hiện như sau:

```
#include < stdio.h>
#include < conio.h>
```

```
#include < stdlib.h>
#define MAX 100
#define TRUE 1
#define FALSE 0
int n, C[MAX], k, count, Stop;
void Init(void) {
    printf("\n Nhập n = "); scanf("%d", &n);
    k = 1; count = 0; C[k] = n;
}
void Result(void) {
    int i; count ++;
    printf("\n Cách chia %d:", count);
    for(i = 1; i <= k; i ++)
        printf("%3d", C[i]);
}
void Next_Division(void) {
    int i, j, R, S, D;
    i = k;
    while(i > 0 && C[i] == 1)
        i -- ;
    if(i > 0) {
        C[i] = C[i] - 1;
        D = k - i + 1;
        R = D / C[i];
        S = D % C[i];
        k = i;
        if(R > 0) {
            for(j = i + 1; j <= i + R; j ++)
```

```
            C[j] = C[i];
        k = k + R;
    }
    if(S > 0) {
        k = k + 1; C[k] = S;
    }
  }
  else Stop = TRUE;
}
void Division(void) {
  Stop = FALSE;
  while(!Stop) {
    Result();
    Next_Division();
  }
}
void main(void) {
  clrscr(); Init(); Division(); getch();
}
```

3.4. THUẬT TOÁN QUAY LUI (BACK TRACK)

Phương pháp sinh kế tiếp có thể giải quyết được các bài toán liệt kê khi ta nhận biết được cấu hình đầu tiên của bài toán. Tuy nhiên, không phải cấu hình sinh kế tiếp nào cũng được sinh một cách đơn giản từ cấu hình hiện tại, ngay kể cả việc phát hiện cấu hình ban đầu cũng không phải dễ tìm vì nhiều khi chúng ta phải chứng minh sự tồn tại của cấu hình. Do vậy, thuật toán sinh kế tiếp chỉ giải quyết được những bài toán liệt kê đơn giản. Để giải quyết những bài toán tổ hợp phức tạp, người ta thường dùng thuật toán quay lui (Back Track) sẽ được trình bày dưới đây.

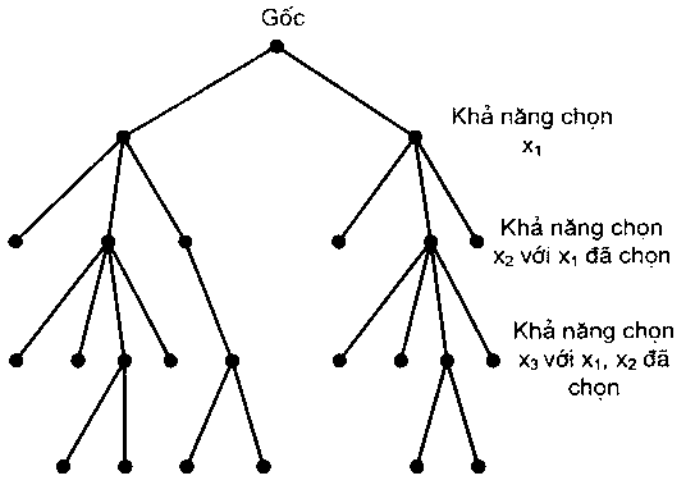
Nội dung chính của thuật toán này là xây dựng dần các thành phần của cấu hình bằng cách thử tất cả các khả năng. Giả sử cần phải tìm một cấu hình của bài toán $x = (x_1, x_2, ..., x_n)$ mà i - 1 thành phần $x_1, x_2, ..., x_{i-1}$ đã được xác định, bây giờ ta xác định thành phần thứ i của cấu hình bằng cách duyệt tất cả các khả năng có thể có và đánh số các khả năng từ 1 ...n_i. Với mỗi khả năng j, kiểm tra xem j có chấp nhận được hay không. Khi đó có thể xảy ra hai trường hợp:

- Nếu chấp nhận j thì xác định x_i theo j, nếu i = n thì ta được một cấu hình cần tìm, ngược lại xác định tiếp thành phần x_{i+1}.
- Nếu thử tất cả các khả năng mà không có khả năng nào được chấp nhận thì quay lại bước trước đó để xác định lại x_{i-1}.

Điểm quan trọng nhất của thuật toán là phải ghi nhớ lại mỗi bước đã đi qua, những khả năng nào đã được thử để tránh sự trùng lặp. Để nhớ lại những bước duyệt trước đó, chương trình cần phải được tổ chức theo cơ chế ngăn xếp (Last in first out). Vì vậy, thuật toán quay lui rất phù hợp với những phép gọi đệ qui. Thuật toán quay lui xác định thành phần thứ i có thể được mô tả bằng thủ tục Try(i) như sau:

```
void Try(int i) {
    int j;
        for(j = 1; j < ni; j ++) {
            if(< Chấp nhận j >) {
                < Xác định xi theo j >
                if(i == n)
                 < Ghi nhận cấu hình > ;
                else Try(i + 1);
            }
        }
}
```

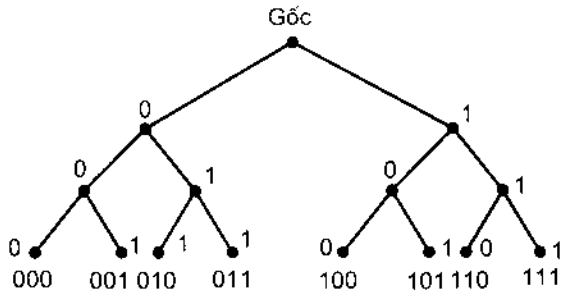
Có thể mô tả quá trình tìm kiếm lời giải theo thuật toán quay lui bằng cây tìm kiếm lời giải sau:



Hình 3.1. Cây liệt kê lời giải theo thuật toán quay lui.

3.4.1. Thuật toán quay lui liệt kê các xâu nhị phân độ dài n

Biểu diễn các xâu nhị phân dưới dạng b_1, b_2, ..., b_n, trong đó $b_i \in \{0, 1\}$. Thủ tục đệ qui Try(i) xác định b_i với các giá trị đề cử cho b_i là 0 và 1. Các giá trị này mặc nhiên được chấp nhận mà không cần phải thoả mãn điều kiện gì (do đó bài toán không cần đến biến trạng thái). Thủ tục Init khởi tạo giá trị n và biến đếm count. Thủ tục kết quả in ra dãy nhị phân tìm được. Chẳng hạn với n = 3, cây tìm kiếm lời giải được thể hiện như hình 3.2.



Hình 3.2. Cây tìm kiếm lời giải liệt kê dãy nhị phân độ dài 3

Chương trình liệt kê các xâu nhị phân có độ dài n sử dụng thuật toán quay lui được thực hiện như sau:

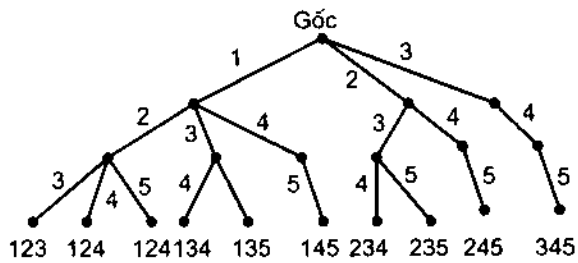
```
#include <stdio.h>
#include <alloc.h>
#include <conio.h>
#include <stdlib.h>
void Result(int *B, int n) {
    int i;
    printf("\n ");
    for(i = 1; i <= n; i ++)
        printf("%3d", B[i]);
}
void Init(int *B, int n) {
    int i;
    for(i = 1; i <= n; i ++)
        B[i] = 0;
}
void Try(int i, int *B, int n) {
    int j;
    for(j = 0; j <= 1; j ++) {
        B[i] = j;
        if(i == n) {
            Result(B, n);
        }
        else Try(i + 1, B, n);
    }
}
void main(void) {
    int *B, n; clrscr();
```

```
    printf("\n Nhap n = "); scanf("%d", &n);
    B = (int *) malloc(n*sizeof(int));
    Init(B, n); Try(1, B, n); free(B);
    getch();
}
```

3.4.2. Thuật toán quay lui liệt kê các tập con m phần tử của tập n phần tử

Biểu diễn tập con k phần tử dưới dạng $c_1, c_2, ..., c_k$, trong đó $1 < c_1 < c_2 ... \leq n$. Từ đó suy ra các giá trị đề cử cho c_i là từ $c_{i-1} + 1$ cho đến n - k + i. Cần thêm vào $c_0 = 0$. Các giá trị đề cử này mặc nhiên được chấp nhận mà không cần phải thêm điều kiện gì. Các thủ tục Init, Result được xây dựng như những ví dụ trên.

Cây tìm kiếm lời giải bài toán liệt kê tập con k phần tử của tập n phần tử với n = 5, k = 3 được thể hiện như trong hình 3.3.



Hình 3.3. Cây liệt kê tổ hợp chập 3 từ {1, 2, 3, 4, 5}

Chương trình liệt kê các tập con k phần tử trong tập n phần tử được thể hiện như sau:

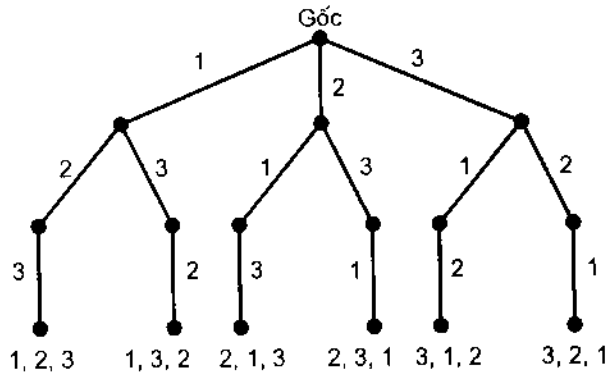
```
#include <conio.h>
#include <stdio.h>
#include <stdlib.h>
#define MAX 100
int B[MAX], n, k, count = 0;
```

```
void Init(void) {
    printf("\n Nhập n = "); scanf("%d", &n);
    printf("\n Nhập k = "); scanf("%d", &k);
    B[0] = 0;
}
void Result(void) {
    int i; count ++;
    printf("\n Tập thứ %d:", count);
    for(i = 1; i <= k; i ++) {
        printf("%3d", B[i]);
    }
    getch();
}
void Try(int i) {
    int j;
    for(j = B[i - 1] + 1; j <= (n - k + i); j ++) {
        B[i] = j;
        if(i == k) Result();
        else Try(i + 1);
    }
}
void main(void) {
    clrscr(); Init(); Try(1);
}
```

3.4.3. Thuật toán quay lui liệt kê các hoán vị của tập n phần tử

Biểu diễn hoán vị dưới dạng $p_1, p_2, .., p_n$, trong đó p_i nhận giá trị từ 1 đến n và $p_i \neq p_j$ với $i \neq j$. Các giá trị từ 1 đến n lần lượt được đề cử cho p_i, trong đó giá trị j được chấp nhận nếu nó chưa được dùng. Vì vậy, cần phải ghi nhớ với mỗi giá trị j xem nó đã được dùng hay chưa.

Điều này được thực hiện nhờ một dãy các biến logic b_j, trong đó b_j = TRUE nếu j chưa được dùng. Các biến này phải được khởi đầu giá trị TRUE trong thủ tục Init. Sau khi gán j cho p_i, cần ghi nhận false cho b_j và phải gán true khi thực hiện xong Result hay Try(i + 1). Các thủ tục còn lại giống như ví dụ 1, 2. Hình 3.4 mô tả cây tìm kiếm lời giải bài toán liệt kê hoán vị của 1, 2, ..., n với n = 3.



Hình 3.4. Cây tìm kiếm lời giải bài toán liệt kê hoán vị của {1, 2, 3}

Sau đây là chương trình giải quyết bài toán liệt kê các hoán vị của 1, 2, ..., n.

```
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
#define MAX 100
#define TRUE 1
#define FALSE 0
int P[MAX], B[MAX], n, count = 0;
void Init(void) {
    int i;
    printf("\n Nhập n = "); scanf("%d", &n);
    for(i = 1; i <= n; i ++)
```

```
        B[i] = TRUE;
}
void Result(void) {
    int i; count ++;
    printf("\n Hoán vị thứ %d:", count);
    for(i = 1; i <= n; i ++)
        printf("%3d", P[i]);
    getch();
}
void Try(int i) {
    int j;
    for(j = 1; j <= n; j ++) {
        if(B[j]) {
            P[i] = j;
            B[j] = FALSE;
            if(i == n) Result();
            else Try(i + 1);
            B[j] = TRUE;
        }
    }
}
void main(void) {
    Init(); Try(1);
}
```

3.4.4. Bài toán Xếp Hậu

Bài toán: Liệt kê tất cả các cách xếp n quân hậu trên bàn cờ n x n sao cho chúng không ăn được nhau.

Bàn cờ có n hàng được đánh số từ 0 đến n - 1, n cột được đánh số từ 0 đến n - 1; Bàn cờ có n*2 - 1 đường chéo xuôi được đánh số từ 0 đến 2*n - 2, 2*n - 1 đường chéo ngược được đánh số từ 2*n - 2. Ví dụ: với bàn cờ 8 x 8, chúng ta có 8 hàng được đánh số từ 0 đến 7, 8 cột được đánh số từ 0 đến 7, 15 đường chéo xuôi, 15 đường chéo ngược được đánh số từ 0 ... 14.

Vì trên mỗi hàng chỉ xếp được đúng một quân hậu, nên chúng ta chỉ cần quan tâm đến quân hậu được xếp ở cột nào. Từ đó dẫn đến việc xác định bộ n thành phần $x_1, x_2, ..., x_n$, trong đó $x_i = j$ được hiểu là quân hậu tại dòng i xếp vào cột thứ j. Giá trị của i được nhận từ 0 đến n - 1; giá trị của j cũng được nhận từ 0 đến n - 1, nhưng thoả mãn điều kiện ô (i, j) chưa bị quân hậu khác chiếu đến theo cột, đường chéo xuôi, đường chéo ngược.

Việc kiểm soát theo hàng ngang là không cần thiết vì trên mỗi hàng chỉ xếp đúng một quân hậu. Việc kiểm soát theo cột được ghi nhận nhờ dãy biến logic a_j với qui ước $a_j = 1$ nếu cột j còn trống, cột $a_j = 0$ nếu cột j không còn trống. Để ghi nhận đường chéo xuôi và đường chéo ngược có chiếu tới ô (i, j) hay không, ta sử dụng phương trình i + j = const và i - j = const, đường chéo thứ nhất được ghi nhận bởi dãy biến b_j, đường chéo thứ 2 được ghi nhận bởi dãy biến c_j với qui ước nếu đường chéo nào còn trống thì giá trị tương ứng của nó là 1 ngược lại là 0. Như vậy, cột j được chấp nhận khi cả 3 biến a_j, b_{i+j}, c_{i+j} đều có giá trị 1. Các biến này phải được khởi đầu giá trị 1 trước đó, gán lại giá trị 0 khi xếp xong quân hậu thứ i và trả lại giá trị 1 khi đưa ra kết quả.

```
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#include <dos.h>
#define N    8
#define D    (2*N - 1)
```

```
#define SG  (N - 1)
#define TRUE  1
#define FALSE 0
void hoanghau(int);
void inloigiai(int loigiai[]); FILE *fp;
int A[N], B[D], C[D], loigiai[N];
int soloigiai = 0;
void hoanghau(int i) {
    int j;
    for(j = 0; j < N; j ++) {
        if(A[j] && B[i - j + SG] && C[i + j]) {
            loigiai[i] = j;
            A[j] = FALSE;
            B[i - j + SG] = FALSE;
            C[i + j] = FALSE;
            if(i == N - 1) {
                soloigiai ++;
                inloigiai(loigiai);
                delay(500);
            }
            else
                hoanghau(i + 1);
            A[j] = TRUE;
            B[i - j + SG] = TRUE;
            C[i + j] = TRUE;
        }
    }
}
```

```
void inloigiai(int *loigiai) {
    int j;
        printf("\n Lời giải %3d:", soloigiai);
        fprintf(fp, "\n Lời giải %3d:", soloigiai);
        for(j = 0; j < N; j ++) {
            printf("%3d", loigiai[j]);
         fprintf(fp, "%3d", loigiai[j]);
        }
}
void main(void) {
    int i; clrscr(); fp = fopen("loigiai.txt", "w");
    for(i = 0; i < N; i ++)
        A[i] = TRUE;
    for(i = 0; i < D; i ++) {
        B[i] = TRUE;
        C[i] = TRUE;
    }
    hoanghau(0);
    fclose(fp);
}
```

3.5. BÀI TOÁN TỐI ƯU

Trong nhiều bài toán thực tế, các cấu hình tổ hợp còn được gán một giá trị bằng số đánh giá giá trị sử dụng của cấu hình đối với một mục đích sử dụng cụ thể nào đó. Khi đó xuất hiện bài toán: hãy lựa chọn trong số tất cả các cấu hình tổ hợp cấu hình có giá trị sử dụng tốt nhất. Các bài toán như vậy được gọi là bài toán tối ưu tổ hợp. Chúng ta có thể phát biểu bài toán tối ưu tổ hợp dưới dạng tổng quát như sau:

Tìm cực tiểu (hay cực đại) của phiếm hàm f(x) = min(max) với điều kiện $x \in D$, trong đó D là tập hữu hạn các phần tử.

Hàm f(x) được gọi là hàm mục tiêu của bài toán, mỗi phần tử $x \in D$ được gọi là một phương án, tập D được gọi là tập các phương án của bài toán. Thông thường tập D được mô tả như là tập các cấu hình tổ hợp thỏa mãn một số tính chất cho trước nào đó.

Phương án $x^* \in D$ đem lại giá trị nhỏ nhất (lớn nhất) cho hàm mục tiêu được gọi là phương án tối ưu, khi đó giá trị $f^* = f(x^*)$ được gọi là giá trị tối ưu của bài toán. Dưới đây chúng ta sẽ giới thiệu một số mô hình tối ưu hóa tổ hợp truyền thống. Các mô hình này là những mô hình có nhiều ứng dụng thực tế, đồng thời chúng giữ vai trò quan trọng trong việc nghiên cứu và phát triển lý thuyết tối ưu hoá tổ hợp.

Bài toán "Người du lịch": Một người du lịch muốn đi thăm quan n thành phố $T_1, T_2, ..., T_n$. Xuất phát từ một thành phố nào đó, người du lịch muốn đi qua tất cả các thành phố còn lại, mỗi thành phố đi qua đúng một lần, rồi quay trở lại thành phố xuất phát. Biết c_{ij} là chi phí đi từ thành phố T_i đến thành phố T_j (i = 1, 2, ..., n), hãy tìm hành trình với tổng chi phí là nhỏ nhất (một hành trình là một cách đi thoả mãn điều kiện).

Rõ ràng, ta có thể thiết lập được một tương ứng 1 - 1 giữa hành trình $T_{\pi(1)} \to T_{\pi(2)} \to ...T_{\pi(n)} \to T_{\pi(1)}$) với một hoán vị $\pi = (\pi(1), \pi(2), ..., \pi(n))$ của n số tự nhiên 1, 2, ..., n. Đặt

$$f(\pi) = C_{\pi(1),\pi(2)} + C_{\pi(2),\pi(3)} + ... + C_{\pi(n-1),\pi(n)} + C_{\pi(n),\pi(1)}$$

Kí hiệu $\prod$ là tập tất cả các hoán vị $\pi = (\pi(1), \pi(2), ..., \pi(n))$ của n số tự nhiên 1, 2, ..., n. Khi đó bài toán người du lịch có thể phát biểu dưới dạng bài toán tối ưu tổ hợp sau:

$$\min \{f(\pi) : \pi \in \prod\}$$

Có thể thấy rằng, tổng số hành trình của người du lịch là n!, trong đó chỉ có (n - 1)! hành trình thực sự khác nhau (bởi vì có thể xuất phát từ một thành phố bất kỳ nên có thể cố định một thành phố nào đó làm điểm xuất phát).

Bài toán "Cái túi": Một nhà thám hiểm cần đem theo một cái túi có trọng lượng không quá b. Có n đồ vật có thể đem theo. Đồ vật thứ j có trọng lượng a_j và giá trị sử dụng c_j (j = 1, 2, ..., n). Hỏi nhà thám hiểm cần đem theo những đồ vật nào để cho tổng giá trị sử dụng là lớn nhất?

Một phương án của nhà thám hiểm có thể biểu diễn như một véc tơ nhị phân độ dài n: $x = (x_1, x_2, ..., x_n)$, trong đó $x_i = 1$ có nghĩa là đồ vật thứ i được đem theo, $x_i = 0$ có nghĩa đồ vật không được đem theo. Với phương án đem theo x, giá trị sử dụng các đồ vật đem theo là

$f(x) = \sum_{i=1}^{n} c_i x_i$, tổng trọng lượng đồ vật đem theo là $g(x) = \sum_{i=1}^{n} a_i x_i$,

như vậy bài toán cái túi được phát biểu dưới dạng bài toán tối ưu tổ hợp sau:

Trong số các vetor nhị phân, độ dài n thoả mãn điều kiện $g(x) \le b$, hãy tìm véc tơ x* cho giá trị lớn nhất của hàm mục tiêu f(x). Nói cách khác:

$$\text{Max } \{f(x): g(x) \le b\}$$

Bài toán "Cho thuê máy": Một ông chủ có một cái máy để cho thuê. Đầu tháng ông ta nhận được yêu cầu thuê máy của m khách hàng. Mỗi khách hàng i sẽ cho biết tập N_i các ngày trong tháng cần sử dụng máy (i = 1, 2, ..., m). Ông chủ chỉ có quyền hoặc từ chối yêu cầu của khách hàng i, hoặc nếu nhận thì phải bố trí máy phục vụ khách hàng i đúng những ngày mà khách hàng này yêu cầu. Hỏi rằng ông chủ phải tiếp nhận các yêu cầu của khách thế nào để cho tổng số ngày sử dụng máy là lớn nhất.

Ký hiệu, I = {1, 2, ..., m} là tập chỉ số khách hàng, S là tập hợp các tập con của I. Khi đó, tập hợp tất cả các phương án cho thuê máy là: $D = \{J \subset S : N_k \cap N_p = \phi, \forall k \neq p \in J\}$. Với mỗi phương án $J \in D$ $f(j) = \sum_{j \in J} |N_j|$ sẽ là tổng số ngày sử dụng máy theo phương án đó. Bài toán đặt ra có thể phát biểu dưới dạng bài toán tối ưu tổ hợp sau:

$$\max\{f(j) : j \in D\}$$

Bài toán "Phân công": Có n công việc và n thợ. Biết c_{ij} là chi phí cần trả để thợ i hoàn thành công việc thứ j (i, j = 1, 2, ..., n). Cần phải thuê thợ sao cho các công việc đều hoàn thành và mỗi thợ chỉ thực hiện một công việc, mỗi công việc chỉ do một thợ thực hiện. Hãy tìm cách thuê n nhân công sao cho tổng chi phí thuê thợ là nhỏ nhất.

Rõ ràng, mỗi phương án bố trí thợ thực hiện các công việc tương ứng với một hoán vị $\pi = (\pi(1), \pi(2), ..., \pi(n))$ của n số tự nhiên {1, 2, ..., n}. Chi phí theo phương án trên là: $f(\pi) = C_{\pi(1),1} + C_{\pi(2),2} + ... + C_{\pi(n),n}$.

Công việc	Thợ thực hiện
1	π(1)
2	π(2)
...	...
n	π(n)

Bài toán đặt ra được dẫn về bài toán tối ưu tổ hợp: $\min\{f(\pi) : \pi \in \Pi\}$.

Bài toán "Lập lịch": Mỗi một chi tiết trong số n chi tiết $D_1, D_2, ..., D_n$ cần phải lần lượt được gia công trên m máy $M_1, M_2, ..., M_m$. Thời gian gia công chi tiết D_i trên mãy M_j là t_{ij}. Hãy tìm lịch (trình tự gia công) các chi tiết trên các mãy sao cho việc hoàn thành gia công tất cả các chi tiết là sớm nhất có thể được. Biết rằng, các chi tiết được gia công một cách liên tục. Nghĩa là quá trình gia công của mỗi một chi tiết phải

được tiến hành một cách liên tục hết máy này sang máy khác, không cho phép có khoảng thời gian dừng khi chuyển từ các máy khác nhau.

Rõ ràng, mỗi một lịch gia công các chi tiết trên các máy sẽ tương ứng với một hoán vị $\pi = (\pi(1), \pi(2), ..., \pi(n))$ của n số tự nhiên 1, 2, ..., n. Thời gian hoàn thành theo các lịch trên được xác định bởi hàm số

$f(\pi) = \sum_{j=1}^{n-1} C_{\pi(j),\pi(j+1)} + \sum_{k=1}^{m} t_{k,\pi(n)}$, trong đó $c_{ij} = S_j - S_i$, S_j là thời điểm bắt đầu thực hiện việc gia công chi tiết j (i, j = 1, 2, ..., n). Ý nghĩa của hệ số c_{ij} có thể được giải thích như sau: nó là tổng thời gian gián đoạn (được tính từ khi bắt đầu gia công chi tiết i) gây ra bởi chi tiết j khi nó được gia công sau chi tiết i trong lịch gia công. Vì vậy, c_{ij} có thể tính theo công thức:

$c_{ij} = \max_{1 \le k \le m}\left[\sum_{l=1}^{k} t_{ij} - \sum_{l=1}^{k-1} t_{lj}\right]$, i, j = 1, 2, ..., n. Vì vậy bài toán đặt ra dẫn về bài toán tối ưu tổ hợp sau: $\min \{f(\pi): \pi \in \Pi\}$

Trong thực tế, lịch gia công còn phải thỏa mãn thêm nhiều điều kiện khác nữa. Vì những ứng dụng quan trọng của những bài toán loại này mà trong tối ưu hoá tổ hợp đã hình thành một lĩnh vực lý thuyết riêng về các bài toán lập lịch, đó là lý thuyết lập lịch hay qui hoạch lịch.

3.6. THUẬT TOÁN DUYỆT

Một trong những phương pháp hiển nhiên nhất để giải bài toán tối ưu tổ hợp đặt ra là duyệt toàn bộ. Trên cơ sở các thuật toán liệt kê tổ hợp, ta tiến hành duyệt từng phương án của bài toán. Đối với mỗi phương án, ta đều tính giá trị hàm mục tiêu tại nó, sau đó so sánh giá trị của hàm mục tiêu tại tất cả các phương án đã được liệt kê để tìm ra phương án tối ưu. Phương pháp xây dựng theo nguyên tắc như vậy được gọi là phương pháp duyệt toàn bộ. Hạn chế của phương pháp duyệt toàn bộ là sự bùng nổ của các cấu hình tổ hợp. Chẳng hạn, để

duyệt được 15! = 1.307.674.368.000 cấu hình, trên máy có tốc độ 1 tỷ phép tính giây, nếu mỗi hoán vị cần liệt kê mất khoảng 100 phép tính, ta cần khoảng thời gian là 130767 giây (lớn hơn 36 tiếng đồng hồ). Vì vậy, cần phải có biện pháp hạn chế việc kiểm tra hoặc tìm kiếm trên các cấu hình tổ hợp thì mới có hy vọng giải được các bài toán tối ưu tổ hợp thực tế. Tất nhiên, để đưa ra được một thuật toán, cần phải nghiên cứu kỹ tính chất của mỗi bài toán tổ hợp cụ thể. Chính nhờ những nghiên cứu đó, trong một số trường hợp ta có thể xây dựng được thuật toán hiệu quả để giải quyết bài toán đặt ra. Nhưng cũng cần phải chú ý rằng, trong nhiều trường hợp (bài toán người du lịch, bài toán cái túi, bài toán cho thuê máy) chúng ta vẫn chưa tìm ra được một phương pháp hữu hiệu nào ngoài phương pháp duyệt toàn bộ đã được đề cập ở trên. Ví dụ sau đây là một điển hình cho phương pháp duyệt toàn bộ.

Ví dụ: Duyệt mọi bộ giá trị trong tập các giá trị rời rạc

Trong thực tế rất thường hay gặp bài toán tối ưu tổ hợp được cho dưới dạng sau:

Tìm: $\max\{f(x_1, x_2, ..., x_n) : x_i \in D_i; i = 1, 2, ..., n\}$

hoặc $\min\{f(x_1, x_2, ..., x_n) : x_i \in D_i; i = 1, 2, ..., n\}$.

Trong đó, D_i là một tập hữu hạn các giá trị rời rạc thỏa mãn một điều kiện ràng buộc nào đó.

Giả sử số các phần tử của tập giá trị rời rạc D_i là r_i (i = 1, 2, ..., n). Gọi $R = r_1 + r_2 + ... + r_n$ là số các phần tử thuộc tất cả các tập D_i (i = 1, 2, ..., n). Khi đó, ta có tất cả C(R, n) bộ có thứ tự các giá trị gồm n phần tử trong R phần tử. Trong số C(R, n) các bộ n phần tử, ta cần lọc ra các bộ thoả mãn điều kiện $x_i \in D_i$ (i = 1, 2, ..., n) để tính giá trị của hàm mục tiêu $f(x_1, x_2, ..., x_n)$. Như vậy, bài toán được đưa về bài toán duyệt các bộ gồm n phần tử $(x_1, x_2, ..., x_n)$ từ tập hợp gồm $R = r_1 + r_2 + ... + r_n$ phần tử thoả mãn điều kiện $x_i \in D_i$.

Ví dụ: Với tập D_1 = (1, 2, 3),

D2 = (3, 4),

D3 = (5, 6, 7).

Khi đó chúng ta cần duyệt bộ các giá trị rời rạc sau:

1	3	5	2	4	5
1	3	6	2	4	6
1	3	7	2	4	7
1	4	5	3	3	5
1	4	6	3	3	6
1	4	7	3	3	7
2	3	5	3	4	5
2	3	6	3	4	6
2	3	7	3	4	7

Chương trình được thể hiện như sau:

```
#include <stdio.h>
#include <stdlib.h>
#include <alloc.h>
#include <conio.h>
#define MAX 2000000
#define TRUE 1
#define FALSE 0
int n, k, H[100]; float *B; int *C, count = 0, m;
FILE *fp;
void Init(void) {
    int i, j; float x; C[0] = 0; H[0] = 0;
    fp = fopen("roirac.in", "r");
    fscanf(fp, "%d", &n);
    printf("\n Số tập con rời rạc n = %d", n);
    for(i = 1; i <= n; i ++) {
        fscanf(fp, "%d", &H[i]);
        printf("\n Hàng %d có số phần tử là %d", i, H[i]);
    }
```

```
    H[0] = 0;
    for(i = 1; i <= n; i ++) {
        printf("\n");
        for(j = 1; j <= H[i]; j ++) {
            fscanf(fp, "%f", &x);
            B[ ++ k] = x;
        }
    }
    printf("\n B = ");
    for(i = 1; i <= k; i ++) {
        printf("%8.2f", B[i]);
    }
    fclose(fp);
}
int In_Set(int i) {
    int canduoi = 0, cantren = 0, j;
    for(j = 1; j <= i; j ++)
        cantren = cantren + H[j];
    canduoi = cantren - H[j - 1];
    if(C[i] > canduoi && C[i] <= cantren)
        return(TRUE);
    return(FALSE);
}
void Result(void) {
    int i;
    count ++; printf("\n Tập con thứ count = %d:", count);
    for(i = 1; i <= n ; i ++) {
        printf("%8.2f", B[C[i]]);
```

```
    }
}
void Try(int i) {
    int j;
    for(j = C[i - 1] + 1; j <= (k - n + i); j ++) {
        C[i] = j;
        if(In_Set(i)) {
            if(i == n) Result();
            else Try(i + 1);
        }
    }
}
void main(void) {
    clrscr();
    B = (float *) malloc(MAX *sizeof(float));
    C = (int *) malloc(MAX *sizeof(int));
    Init(); Try(1); free(B); free(C); getch();
}
```

3.7. THUẬT TOÁN NHÁNH CẬN

Giả sử, chúng ta cần giải quyết bài toán tối ưu tổ hợp với mô hình tổng quát như sau:

$$\min\{f(x) : x \in D\}$$

Trong đó, D là tập hữu hạn phần tử. Ta giả thiết D được mô tả như sau:

$D = \{x = (x_1, x_2, ..., x_n) \in A_1 \times A_2 \times ... \times A_n$; x thoả mãn tính chất P$\}$, với $A_1 \times A_2 \times ... \times A_n$ là các tập hữu hạn, P là tính chất cho trên tích đề xác $A_1 \times A_2 \times ... \times A_n$.

Như vậy, các bài toán chúng ta vừa trình bày ở trên đều có thể được mô tả dưới dạng trên.

Với giả thiết về tập D như trên, chúng ta có thể sử dụng thuật toán quay lui để liệt kê các phương án của bài toán. Trong quá trình liệt kê theo thuật toán quay lui, ta sẽ xây dựng dần các thành phần của phương án. Một bộ phận gồm k thành phần (a_1, a_2, ..., a_k) xuất hiện trong quá trình thực hiện thuật toán sẽ được gọi là phương án bộ phận cấp k.

Thuật toán nhánh cận có thể được áp dụng giải bài toán đặt ra, nếu như có thể tìm được một hàm g xác định trên tập tất cả các phương án bộ phận của bài toán thoả mãn bất đẳng thức sau:

$$g(a_1, a_2, ..., a_k) \le \min\{f(x) : x \in D, x_i = a_i, i = 1, 2, ..., k\} \qquad (*)$$

Với mọi lời giải bộ phận (a_1, a_2, ..., a_k), và với mọi k = 1, 2, ...

Bất đẳng thức (*) có nghĩa là giá trị của hàm tại phương án bộ phận (a_1, a_2, ..., a_k) không vượt quá giá trị nhỏ nhất của hàm mục tiêu bài toán trên tập con các phương án.

$D(a_1, a_2, ..., a_k)$ $\{x \in D: x_i = a_i, 1 = 1, 2, ..., k\}$,

Nói cách khác, $g(a_1, a_2, .., a_k)$ là cận dưới của tập $D(a_1, a_2, ..., a_k)$. Do có thể đồng nhất tập $D(a_1, a_2, ..., a_k)$ với phương án bộ phận (a_1, a_2, ..., a_k), nên ta cũng gọi giá trị $g(a_1, a_2, ..., a_k)$ là cận dưới của phương án bộ phận (a_1, a_2, ..., a_k).

Giả sử, ta đã có được hàm g. Ta xét cách sử dụng hàm này để hạn chế khối lượng duyệt trong quá trình duyệt tất cả các phương án theo thuật toán quay lui. Trong quá trình liệt kê các phương án, có thể đã thu được một số phương án của bài toán. Gọi $\bar{x}$ là giá trị hàm mục tiêu nhỏ nhất trong số các phương án đã duyệt, ký hiệu $\bar{f} = f(\bar{x})$. Ta gọi $\bar{x}$ là phương án tốt nhất hiện có, còn $\bar{f}$ là kỷ lục. Giả sử, ta có được $\bar{f}$, khi đó nếu:

$g(a_1, a_2, .., a_k) > \bar{f}$ thì từ bất đẳng thức (*) ta suy ra

$\bar{f} < g(a_1, a_2, ..., a_k) \leq \min \{f(x): x \in D, x_i = a_i, i = 1, 2, ..., k\}$.

Vì thế tập con các phương án của bài toán $D(a_1, a_2, ..., a_k)$ chắc chắn không chứa phương án tối ưu. Trong trường hợp này, ta không cần phải phát triển phương án bộ phận $(a_1, a_2, ..., a_k)$. Nói cách khác, ta có thể loại bỏ các phương án trong tập $D(a_1, a_2, ..., a_n)$ khỏi quá trình tìm kiếm.

Thuật toán quay lui liệt kê các phương án cần sửa đổi lại như sau:

```
Procedure Try(k);
(*Phát triển phương án bộ phận (a1, a2, ..., ak-1) theo thuật toán quay lui có
kiểm tra cận dưới trước khi tiếp tục phát triển phương án*)
begin
  for ak ∈ Ak do
  begin
     if < chấp nhận ak > then
     begin
        xk : = ak;
        if k = n then < cập nhật kỷ lục >
        else if g(a1, a2, ..., ak) ≤ f̄ then Try (k + 1);
     end;
  end;
end;
```

Khi đó, thuật toán nhánh cận được thực hiện nhờ thủ tục sau:

```
Procedure Nhanh_Can;
Begin
  f̄ = + ∞;
(* Nếu biết một phương án x̄ nào đó thì có thể đặt f̄ = f(x̄) *)
```

```
    Try(1);
    if f̄ ≤ + ∞ then < f̄ là giá trị tối ưu, x̄ là phương án tối ưu >
    else < bài toán không có phương án >;
  end;
```

Chú ý rằng, nếu trong thủ tục Try ta thay thế câu lệnh:

```
    if k = n then < cập nhật kỷ lục >
    else if g(a1, a2, ..., ak) ≤ f̄ then Try(k + 1);
```

Bởi:

```
    if k = n then < cập nhật kỷ lục >
    else Try(k + 1);
```

Thì thủ tục Try sẽ liệt kê toàn bộ các phương án của bài toán, và ta lại thu được thuật toán duyệt toàn bộ. Việc xây dựng hàm g phụ thuộc vào từng bài toán tối ưu tổ hợp cụ thể. Nhưng chúng ta cố gắng xây dựng sao cho việc tính giá trị của g phải đơn giản và giá trị của $g(a_1, a_2, ..., a_k)$ phải sát với giá trị của hàm mục tiêu.

3.7.1. Thuật toán nhánh cận giải bài toán cái túi

Chúng ta sẽ xét bài toán cái túi tổng quát hơn mô hình đã được trình bày trong mục trên. Thay vì có n đồ vật, ở đây ta giả thiết rằng có n loại đồ vật và số lượng đồ vật mỗi loại là không hạn chế. Khi đó, ta có mô hình bài toán cái túi biến nguyên sau đây: Có n loại đồ vật, đồ vật thứ j có trọng lượng a_j và giá trị sử dụng c_j (j = 1, 2, ..., n). Cần chất các đồ vật này vào một cái túi có trọng lượng là b sao cho tổng giá trị sử dụng của các đồ vật đựng trong túi là lớn nhất. Mô hình toán học của bài toán có dạng sau:

$$f^* = \max\left\{ f(x) = \sum_{j=1}^{n} c_j x_j : \sum_{j=1}^{n} a_j x_j \le b, x_j \in Z_+, j = 1,2,...,n \right\}, \qquad (1)$$

Trong đó, Z+ là tập các số nguyên không âm.

Ký hiệu D là tập các phương án của bài toán (1):

$$D = \left\{ x = (x_1, x_2, ..., x_n : \sum_{j=1}^{n} a_j x_j \le b, x_j \in Z_+, j = 1,2,...,n \right\}$$

Không giảm tính tổng quát, ta giả thiết rằng, các đồ vật được đánh số sao cho bất đẳng thức sau được thoả mãn:

$$\frac{c_1}{a_1} \ge \frac{c_2}{a_2} \ge ... \ge \frac{c_n}{a_n} \qquad (2)$$

Để xây dựng hàm tính cận dưới, cùng với bài toán cái túi (1) ta xét bài toán cái túi biến liên tục sau:

$$\text{Tìm } g^* = \max\left\{ \sum_{j=1}^{n} c_j x_j : \sum_{j=1}^{n} a_j x_j \le b, x_j \ge 0, j = 1,2,...,n \right\}. \qquad (3)$$

Người ta đã chứng minh được rằng phương án tối ưu của bài toán (3) là véc tơ $\overline{x} = (\overline{x_1}, \overline{x_2}, ..., \overline{x_n})$ với các thành phần được xác định bởi công thức:

$$\overline{x_1} = \frac{b}{a_1}, \overline{x_2} = \overline{x_3} = ... = \overline{x_n} = 0$$

Và giá trị tối ưu là $g^* = \frac{c_1 b_1}{a_1}$.

Bây giờ, giả sử có phương án bộ phận cấp k: (u_1, u_2, ..., u_k). Khi đó, giá trị sử dụng của các đồ vật đang có trong túi là:

$$\partial_k = c_1 u_1 + c_2 u_2 + ... + c_k u_k,$$

Và trọng lượng còn lại của túi là:

$$b_k = b - c_1 u_1 + c_2 u_2 + ... + c_k u_k,$$

Ta có:

$$\max\{f(x) : x \in D, x_j = u_j, j = 1,2....,n\}$$

$$= \max\left\{\partial_k + \sum_{j=k+1}^{n} c_j x_j : \sum_{j=k+1}^{n} a_j x_j \le b_k, x_j \in Z_+, j = k+1, k+2,..., n\right\}$$

$$\le \partial_k + \max\left\{\sum_{j=k+1}^{n} c_j x_j : \sum_{j=k+1}^{n} a_j x_j \le b_k, x_j \ge 0, j = k+1, k+2,..., n\right\}$$

$$= \partial_k + \frac{c_{k+1} b_k}{a_{k+1}}$$

(Như vậy giá trị số hạng thứ hai là $c_{k+1} b_k / a_{k+1}$)

Từ đó ta có thể tính cận trên cho phương án bộ phận (u_1, u_2, ..., u_k) theo công thức:

$$g(u_1, u_2,..., u_k) = \partial_k + \frac{c_{k+1} b_k}{a_{k+1}}$$

Chú ý: Khi tiếp tục xây dựng thành phần thứ k + 1 của lời giải, các giá trị đề cử cho x_{k+1} sẽ là 0, 1, ..., $[b_k/a_{k+1}]$. Do có kết quả của mệnh đề, khi chọn giá trị cho x_{k+1} ta sẽ duyệt các giá trị đề cử theo thứ tự giảm dần. Chẳng hạn, giải bài toán cái túi sau theo thuật toán nhánh cận trình bày:

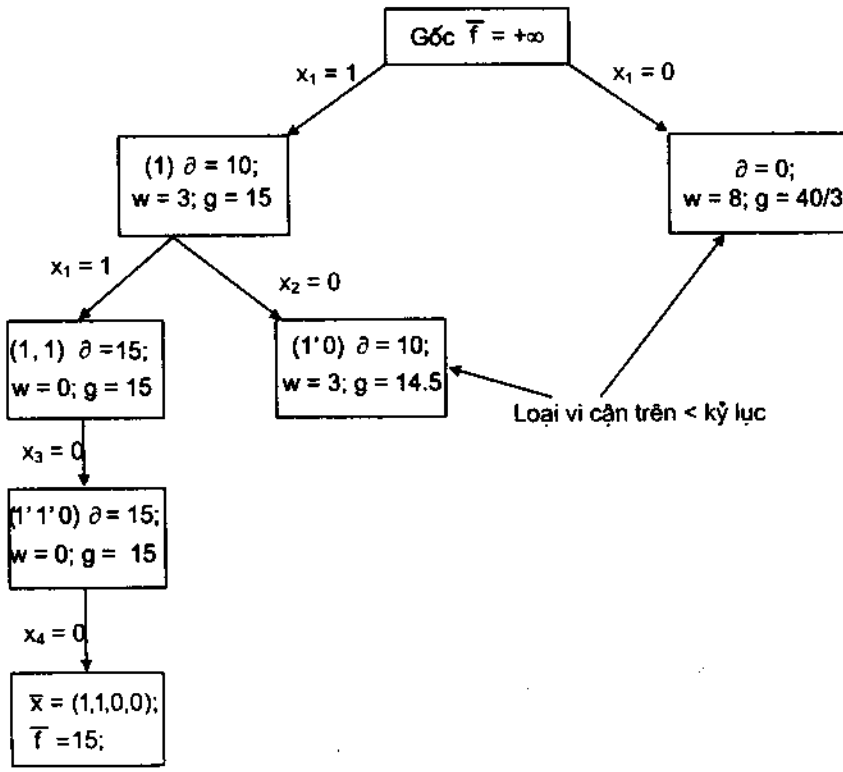
$$f(x) = 10x_1 + 5x_2 + 3x_3 + 6x_4 \to \max$$

$$5x_1 + 3x_2 + 2x_3 + 4x_4 \le 8$$

$$x_j \in Z_+, j = 1,2,3,4.$$

Quá trình giải bài toán được mô tả trong cây tìm kiếm trong hình 3.5. Thông tin về một phương án bộ phận trên cây được ghi trong các ô trên hình vẽ tương ứng theo thứ tự sau:

- Đầu tiên là các thành phần của phương án
- Tiếp đến ∂ là giá trị của các đồ vật chất trong túi
- u là trọng lượng còn lại của túi
- g là cận trên.



Hình 3.5. Giải bài toán cái túi theo thuật toán nhánh cận.

Kết thúc thuật toán, ta thu được phương án tối ưu là:

$$x^* = (1, 1, 0, 1),$$

Giá trị tối ưu $f^* = 15$.

Chương trình giải bài toán cái túi theo thuật toán nhánh cận được thể hiện như sau:

```
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
#include <math.h>
#include <dos.h>
#define TRUE 1
```

```
#define FALSE 0
#define MAX 100
int x[MAX], xopt[MAX];
float fopt, cost, weight;
void Init(float *C, float *A, int *n, float *w) {
    int i; FILE *fp;
    fopt = 0; weight = 0;
    fp = fopen("caitui.in", "r");
    if(fp == NULL) {
        printf("\n Không có tệp input");
        delay(2000); return;
    }
    fscanf(fp, "%d %f", n, w);
    for(i = 1; i <= *n; i ++) xopt[i] = 0;
    printf("\n Số lượng đồ vật %d:", *n);
    printf("\n Giới hạn túi %8.2f:", *w);
    printf("\n Véc tơ giá trị:");
    for(i = 1; i <= *n; i ++) {
        fscanf(fp, "%f", &C[i]);
        printf("%8.2f", C[i]);
    }
    printf("\n Véc tơ trọng lượng:");
    for(i = 1; i <= *n; i ++) {
        fscanf(fp, "%f", &A[i]);
        printf("%8.2f", A[i]);
    }
    fclose(fp);
}
```

```
void swap(int n) {
    int i;
    for(i = 1; i <= n; i ++)
        xopt[i] = x[i];
}
void Update_Kyluc(int n) {
    if(cost > fopt) {
        swap(n);
        fopt = cost;
    }
}
void Try(float *A, float *C, int n, float w, int i) {
    int j, t = (w - weight)/A[i];
    for(j = t; j > = 0; j --) {
        x[i] = j;
        cost = cost + C[i]*x[i];
        weight = weight + x[i]*A[i];
        if(i == n) Update_Kyluc(n);
        else if(cost + C[i + 1]*(w - weight)/A[i + 1] > fopt) {
            Try(A, C, n, w, i + 1);
        }
        weight = weight - A[i]*x[i];
        cost = cost - C[i]*x[i];
    }
}
void Result(int n) {
    int i;
    printf("\n Giá trị đồ vật %8.2f:", fopt);
```

```
    printf("\n Phương án tối ưu:");
    for(i = 1; i <= n; i ++)
        printf("%3d", xopt[i]);
}
void main(void) {
    int n;
    float A[MAX], C[MAX], w;
    clrscr(); Init(C, A, &n, &w);
    Try(C, A, n, w, 1); Result(n);
    getch();
}
```

3.7.2. Thuật toán nhánh cận giải bài toán người du lịch

Bài toán "Người du lịch". Một người du lịch muốn đi thăm quan n thành phố T_1, T_2, ... , T_n. Xuất phát từ một thành phố nào đó, người du lịch muốn đi qua tất cả các thành phố còn lại, mỗi thành phố đi qua đúng một lần, rồi quay trở lại thành phố xuất phát. Biết c_{ij} là chi phí đi từ thành phố T_i đến thành phố T_j (i, j = 1, 2, ..., n), hãy tìm hành trình với tổng chi phí là nhỏ nhất (một hành trình là một cách đi thoả mãn điều kiện).

Giải: Cố định thành phố xuất phát là T_1. Bài toán Người du lịch được đưa về bài toán: Tìm cực tiểu của phiếm hàm:

$$f(x_1, x_2, ..., x_n) = c[1, x_2] + c[x_2, x_3] + ... + c[x_{n-1}, x_n] + c[x_n, x_1] \rightarrow \min$$

Với điều kiện

$c_{min} = \min\{c[i, j], i, j = 1,2,...,n; i \neq j\}$ là chi phí đi lại giữa các thành phố.

Giả sử ta đang có phương án bộ phận (u_1, u_2, ..., u_k). Phương án tương ứng với hành trình bộ phận qua k thành phố:

$$T_1 \rightarrow T(u_2) \rightarrow ... \rightarrow T(u_{k-1}) \rightarrow T(u_k)$$

Vì vậy, chi phí phải trả theo hành trình bộ phận này sẽ là tổng các chi phí theo từng node của hành trình bộ phận.

$$\partial = c[1,u_2] + c[u_2,u_3] + ... + c[u_{k-1}, u_k] .$$

Để phát triển hành trình bộ phận này thành hành trình đầy đủ, ta còn phải đi qua n - k thành phố còn lại rồi quay trở về thành phố T_1, tức là còn phải đi qua n - k + 1 đoạn đường nữa. Do chi phí phải trả cho việc đi qua mỗi trong n - k + 1 đoạn đường còn lại đều không nhiều hơn c_{min}, nên cận dưới cho phương án bộ phận $(u_1, u_2, ..., u_k)$ có thể được tính theo công thức:

$$g(u_1, u_2, ..., u_k) = \partial + (n - k + 1) * c_{min}$$

Chẳng hạn, giải bài toán người du lịch với ma trận chi phí như sau

$$C = \begin{vmatrix} 0 & 3 & 14 & 18 & 15 \\ 3 & 0 & 4 & 22 & 20 \\ 17 & 9 & 0 & 16 & 4 \\ 6 & 2 & 7 & 0 & 12 \\ 9 & 15 & 11 & 5 & 0 \end{vmatrix}$$

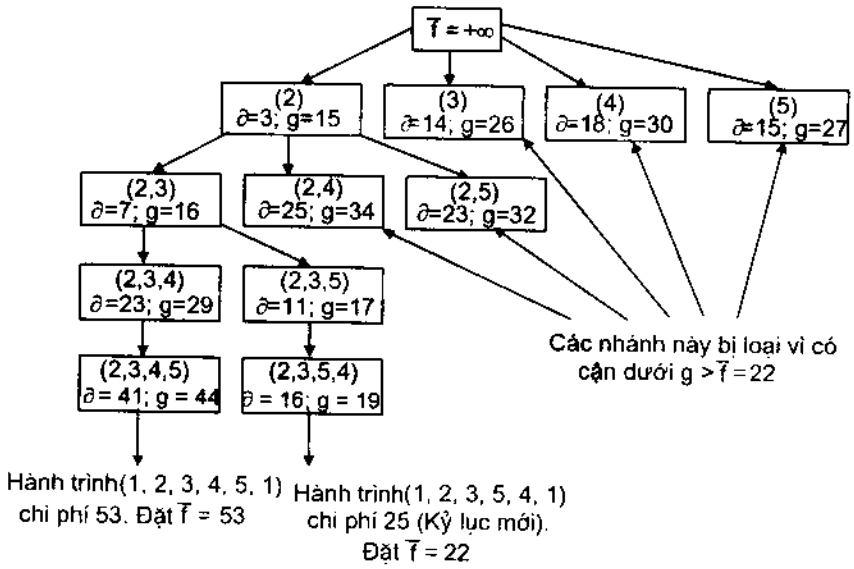
Ta có $c_{min} = 2$. Quá trình thực hiện thuật toán được mô tả bởi cây tìm kiếm lời giải được thể hiện trong hình 3.6.

Thông tin về một phương án bộ phận trên cây được ghi trong các ô trên hình vẽ tương ứng theo thứ tự sau:

- Đầu tiên là các thành phần của phương án
- Tiếp đến ∂ là chi phí theo hành trình bộ phận
- g là cận dưới

Kết thúc thuật toán, ta thu được phương án tối ưu (1, 2, 3, 5, 4, 1) tương ứng với phương án tối ưu với hành trình:

$T_1 \rightarrow T_2 \rightarrow T_3 \rightarrow T_5 \rightarrow T_4 \rightarrow T_1$ và chi phí nhỏ nhất là 22



Hình 3.6. Cây tìm kiếm lời giải bài toán người du lịch.

Chương trình giải bài toán theo thuật toán nhánh cận được thể hiện như sau:

```
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#include <io.h>
#define MAX 20
int n, P[MAX], B[MAX], C[20][20], count = 0;
int A[MAX], XOPT[MAX];
int can, cmin, fopt;
void Read_Data(void) {
    int i, j; FILE *fp;
    fp = fopen("dulich.in","r");
    fscanf(fp,"%d", &n);
```

```
    printf("\n Số thành phố: %d", n);
    printf("\n Ma trận chi phí:");
    for(i = 1; i<= n; i ++) {
        printf("\n");
        for(j = 1; j<= n; j ++) {
            fscanf(fp,"%d",&C[i][j]);
            printf("%5d", C[i][j]);
        }
    }
}
int Min_Matrix(void) {
    int min = 1000, i, j;
    for(i = 1; i <= n; i ++) {
        for(j = 1; j <= n; j ++) {
            if(i! = j && min>C[i][j])
                min = C[i][j];
        }
    }
    return(min);
}
void Init(void) {
    int i;
    cmin = Min_Matrix();
    fopt = 32000; can = 0; A[1] = 1;
    for(i = 1; i <= n; i ++)
        B[i] = 1;
}
void Result(void) {
```

```
    int i;
    printf("\n Hành trình tối ưu %d:", fopt);
    printf("\n Hành trình:");
    for(i = 1; i<= n; i ++)
        printf("%3d ->", XOPT[i]);
    printf("%d",1);
}
void Swap(void) {
    int i;
    for(i = 1; i<= n; i ++)
        XOPT[i] = A[i];
}
void Update_Kyluc(void) {
    int sum;
    sum = can + C[A[n]][A[1]];
    if(sum < fopt) {
        Swap();
        fopt = sum;
    }
}
void Try(int i) {
    int j;
    for(j = 2; j<= n; j ++) {
        if(B[j]) {
            A[i] = j; B[j] = 0;
            can = can + C[A[i - 1]][A[i]];
            if(i == n) Update_Kyluc();
            else if(can + (n - i + 1)*cmin< fopt) {
```

```
            count ++;
            Try(i + 1);
        }
        B[j] = 1; can = can - C[A[i - 1]][A[i]];
    }
  }
}
void main(void) {
  clrscr(); Read_Data(); Init();
  Try(2); Result();
  getch();
}
```

BÀI TẬP CHƯƠNG 3

3.1. **Duyệt mọi tập con của tập hợp 1, 2, ..., n.** Dữ liệu vào cho bởi tệp tapcon.in, kết quả ghi lại trong tệp bai11.out. Ví dụ sau sẽ minh họa cho tệp tapcon.in và tapcon.out.

tapcon.in	tapcon.out
3	1
	2
	2 1
	3
	3 1
	3 2
	3 2 1

3.2. **Tìm tập con dài nhất có thứ tự tăng dần, giảm dần.** Cho dãy số $a_1, a_2, ..., a_n$. Hãy tìm dãy con dài nhất được sắp xếp theo thứ tự tăng hoặc giảm dần. Dữ liệu vào cho bởi tệp tapcon.in, dòng đầu tiên ghi lại số tự nhiên n ($n \leq 100$), dòng kế tiếp ghi lại n số, mỗi số được phân biệt với nhau bởi một hoặc vài ký tự rỗng. Kết quả ghi lại trong tệp tapcon.out. Ví dụ sau sẽ minh họa cho tệp tapcon.in và tapcon.out.

tapcon.in	tapcon.out
5	5
71 3 8 9 6 12	1 3 8 9 12

3.3. **Duyệt các tập con thoả mãn điều kiện**. Cho dãy số $a_1, a_2, ..., a_n$ và số M. Hãy tìm tất cả các dãy con trong dãy số $a_1, a_2, ..., a_n$ sao cho tổng các phần tử trong dãy con đúng bằng M. Dữ liệu vào cho bởi tệp tapcon.in, dòng đầu tiên ghi lại hai số tự nhiên N và số M ($N \leq 100$), dòng kế tiếp ghi lại N số mỗi số được phân biệt với nhau bởi một và dấu trống. Kết quả ghi lại trong tệp tapcon.out. Ví dụ sau sẽ minh họa cho tệp tapcon.in và tapcon.out

tapcon.in

50

5 10 15 20 25 30 35

tapcon.out

20 30

15 35

10 15 25

5 20 25

5 15 30

5 10 35

5 10 15 20

3.4. Cho lưới hình chữ nhật gồm (n×m) hình vuông đơn vị. Hãy liệt kê tất cả các đường đi từ điểm có tọa độ (0, 0) đến điểm có tọa độ (n×m). Biết rằng, điểm (0, 0) được coi là đỉnh dưới của hình vuông dưới nhất góc bên trái, mỗi bước đi chỉ được phép thực hiện hoặc lên trên hoặc xuống dưới theo cạnh của hình vuông đơn vị. Dữ liệu vào cho bởi tệp bai14.inp, kết quả ghi lại trong tệp bai14.out. Ví dụ sau sẽ minh họa cho tệp bai14.in và bai14.out.

bai14.in

2 2

bai14.out

0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0

3.5. **Duyệt mọi tập con k phần tử từ tập gồm n phần tử.** Dữ liệu vào cho bởi tệp tapcon.in, kết quả ghi lại trong tệp tapcon.out. Ví dụ sau sẽ minh họa cho tapcon.in và tapcon.out.

tapcon.in

5 3

tapcon.out

1	2	3
1	2	4
1	2	5
1	3	4
1	3	5
1	4	5
2	3	4
2	3	5
2	4	5
3	4	5

3.6. Duyệt các tập con k phần tử thỏa mãn điều kiện. Cho dãy số a_1, a_2, ..., a_n và số M. Hãy tìm tất cả các dãy con k phần tử trong

dãy số $a_1, a_2, ..., a_n$ sao cho tổng các phần tử trong dãy con đúng bằng M. Dữ liệu vào cho bởi tệp tapcon.in, dòng đầu tiên ghi lại số tự nhiên n , k và số M, hai số được viết cách nhau bởi một vài ký tự trống, dòng kế tiếp ghi lại n số mỗi số được viết cách nhau bởi một hoặc vài ký tự trống. Kết quả ghi lại trong tệp tapcon.out. Ví dụ sau sẽ minh họa cho tệp tapcon.in và tapcon.out.

tapcon.in

7	3	50				
5	10	15	20	25	30	35

tapcon.out

5	10	35
5	15	35
5	20	25
10	15	25

3.7. Duyệt mọi hoán vị của từ COMPUTER. Dữ liệu vào cho bởi tệp hoanvi.in, kết quả ghi lại trong tệp hoanvi.out.

3.8. Duyệt mọi ma trận các hoán vị. Cho hình vuông gồm $n \times n$ ($n \geq 5$, n lẻ) hình vuông đơn vị. Hãy điền các số từ 1, 2, ..., n vào các hình vuông đơn vị sao cho những điều kiện sau được thoả mãn:

a) Đọc theo hàng ta nhận được n hoán vị khác nhau của 1, 2, ..., n;

b) Đọc theo cột ta nhận được n hoán vị khác nhau của 1, 2, ..., n;

c) Đọc theo hai đường chéo ta nhận được 2 hoán vị khác nhau của 1, 2, ..., n;

Hãy tìm ít nhất 1 (hoặc tất cả) các hình vuông thoả mãn 3 điều kiện trên. Dữ liệu vào cho bởi tệp hoanvi.in, kết quả ghi lại

trong tệp hoanvi.out. Ví dụ sau sẽ minh họa cho tệp input & output của bài toán.

hoanvi.in

5

hoanvi.out

5	3	4	1	2
1	2	5	3	4
3	4	1	2	5
2	5	3	4	1
4	1	2	5	3

3.9. Duyệt mọi cách chia số tự nhiên n thành tổng các số nguyên nhỏ hơn. Dữ liệu vào cho bởi tệp chiaso.in, kết quả ghi lại trong tệp chiaso.out. Ví dụ sau sẽ minh họa cho tệp input & output của bài toán.

chiaso.in

4

chiaso.out

4			
3	1		
2	2		
2	1	1	
1	1	1	1

3.10. **Duyệt mọi bộ giá trị trong tập các giá trị rời rạc.** Cho k tập hợp các số thực $A_1, A_2, ..., A_k (k \leq 10)$ có số các phần tử tương ứng là $N_1, N_2, ..., N_k$ (các tập có thể có những phần tử giống

nhau). Hãy duyệt tất cả các bộ k phần tử $a = (a_1, a_2, ..., a_k)$ sao cho $a_i \in A_i$ $(i = 1, 2, ..., k)$. Dữ liệu vào cho bởi tệp chiaso.in, dòng đầu tiên ghi lại k + 1 số tự nhiên, mỗi số được phân biệt với nhau bởi một vài dấu trống là giá trị của n, $N_1, N_2, ..., N_k$; k dòng kế tiếp ghi lại các phần tử của tập hợp $A_1, A_2, ..., A_k$. Kết quả ghi lại trong tệp chiaso.out, mỗi phần tử được phân biệt với nhau bởi một vài dấu trống. Ví dụ sau sẽ minh họa cho tệp input & output của bài toán.

chiaso.inp

```
3   3   2   2
1   2   3
4   5
6   7
```

chiaso.out

```
1   4   6
1   4   7
1   5   6
1   5   7
2   4   6
2   4   7
2   5   6
2   5   7
3   4   6
3   4   7
3   5   6
3   5   7
```

3.11. Tìm bộ giá trị rời rạc trong bài 21 để hàm mục tiêu $\sin(x_1 + x_2 + ... + x_k)$ đạt giá trị lớn nhất. Dữ liệu vào cho bởi tệp bai22.inp, kết quả ghi lại trong tệp bai22.out.

3.12. Duyệt mọi phép toán trong tính toán giá trị biểu thức. Viết chương trình nhập từ bàn phím hai số nguyên M, N. Hãy tìm cách thay các dấu hỏi (?) trong biểu thức sau bởi các phép toán + , - , *, %, / (chia nguyên) sao cho giá trị của biểu thức nhận được bằng đúng N:

((((M?M) ?M)?M)?M)?M)?M

Nếu không được hãy đưa ra thông báo là không thể được.

3.13. Bài toán cái túi với số lượng đồ vật không hạn chế. Một nhà thám hiểm đem theo một cái túi có trọng lượng không quá b. Có n đồ vật cần đem theo, đồ vật thứ i có trọng lượng tương ứng là một số a_i và giá trị sử dụng c_i ($1 \le i \le n$). Hãy tìm cách bỏ các đồ vật vào túi sao cho tổng giá trị sử dụng các đồ vật là lớn nhất. Biết rằng số lượng các đồ vật là không hạn chế. Dữ liệu vào cho bởi tệp caitui.in. dòng đầu tiên ghi lại số tự nhiên n và số thực b hai số được viết cách nhau bởi một dấu trống, hai dòng kế tiếp ghi n số trên mỗi dòng, tương ứng với vector giá trị sử dụng c_i và vector trọng lượng a_i. Kết quả ghi lại trong tệp caitui.out trên 3 dòng, dòng đầu ghi lại giá trị sử dụng tối ưu, dòng kế tiếp ghi lại loại đồ vật cần đem theo, dòng cuối cùng ghi lại số lượng của mỗi loại đồ vật. Ví dụ sau sẽ minh họa cho tệp input & output của bài toán.

caitui.in

```
8
10    5    3    6
5     3    2    4
```

caitui.out

15

1 1 0 0

1 1 0 0

3.14. Bài toán cái túi với số lượng đồ vật hạn chế. Một nhà thám hiểm đem theo một cái túi có trọng lượng không quá b. Có n đồ vật cần đem theo, đồ vật thứ i có trọng lượng tương ứng là một số a_i và giá trị sử dụng c_i ($1 \le i \le n$). Hãy tìm cách bỏ các đồ vật vào túi sao cho tổng giá trị sử dụng các đồ vật là lớn nhất. Biết rằng số lượng mỗi đồ vật là 1. Dữ liệu vào cho bởi tệp caitui.in, dòng đầu tiên ghi lại số tự nhiên n và số thực b hai số được viết cách nhau bởi một dấu trống, hai dòng kế tiếp ghi n số trên mỗi dòng, tương ứng với vector giá trị sử dụng c_i và vector trọng lượng a_i. Kết quả ghi lại trong tệp caitui.out trên 2 dòng, dòng đầu ghi lại giá trị sử dụng tối ưu, dòng kế tiếp ghi lại loại đồ vật cần đem theo. Ví dụ sau sẽ minh hoạ cho tệp input & output của bài toán.

caitui.in

4 8

8 5 3 1

4 3 2 1

caitui.out

14

1 1 0 1

3.15. Bài toán người du lịch. Một người du lịch muốn đi tham quan tại n thành phố khác nhau. Xuất phát tại một thành phố nào đó, người du lịch muốn đi qua tất cả các thành phố còn lại mỗi thành phố đúng một lần rồi quay trở lại thành phố ban đầu. Biết

C_{ij} là chi phí đi lại từ thành phố thứ i đến thành phố thứ j. Hãy tìm hành trình có chi phí thấp nhất cho người du lịch. Dữ liệu vào cho bởi tệp dulich.in, dòng đầu tiên ghi lại số tự nhiên n, n dòng kế tiếp ghi lại ma trận chi phí C_{ij}. Kết quả ghi lại trong tệp dulich.out, dòng đầu tiên ghi lại chi phí tối ưu, dòng kế tiếp ghi lại hành trình tối ưu. Ví dụ sau sẽ minh họa cho tệp input & output của bài toán.

dulich.in

```
5
00    48    43    54    31
20    00    30    63    22
29    64    00    04    17
06    19    02    00    08
01    28    07    18    00
```

dulich.out

```
81
1    5    3    4    2    1
```

CHƯƠNG 4

NGĂN XẾP, HÀNG ĐỢI VÀ DANH SÁCH MÓC NỐI (STACK, QUEUE, LINK LIST)

4.1. KIỂU DỮ LIỆU NGĂN XẾP VÀ ỨNG DỤNG

4.1.1. Định nghĩa và khai báo

Ngăn xếp (Stack) hay bộ xếp chồng là một kiểu danh sách tuyến tính đặc biệt mà phép bổ sung phần tử và loại bỏ phần tử luôn luôn được thực hiện ở một đầu gọi là đỉnh (top).

Có thể hình dung stack như một chồng đĩa được xếp vào hộp hoặc một băng đạn được nạp vào khẩu súng liên thanh. Quá trình xếp đĩa hoặc nạp đạn chỉ được thực hiện ở một đầu, chiếc đĩa hoặc viên đạn cuối cùng lại chiếm vị trí ở đỉnh đầu tiên còn đĩa đầu hoặc viên đạn đầu lại ở đáy của hộp (bottom), khi lấy ra thì đĩa cuối cùng hoặc viên đạn cuối cùng lại được lấy ra trước tiên. Nguyên tắc vào sau ra trước của stack còn được gọi dưới một tên khác LIFO (Last In First Out).

Stack có thể rỗng hoặc bao gồm một số phần tử. Có hai thao tác chính cho stack là thêm một nút vào đỉnh stack (push) và loại bỏ một nút tại đỉnh stack (pop). Nếu ta muốn thêm một nút vào đỉnh stack thì trước đó ta phải kiểm tra xem stack đã đầy (full) hay chưa, nếu ta muốn loại bỏ một nút của stack thì ta phải kiểm tra stack có rỗng hay không. Hình 4.1 minh họa sự thay đổi của stack thông qua các thao tác thêm và bớt đỉnh trong stack.

Giả sử ta có một stack S lưu trữ các kí tự. Trạng thái bắt đầu của stack được mô tả trong hình 4.1a. Khi đó thao tác:

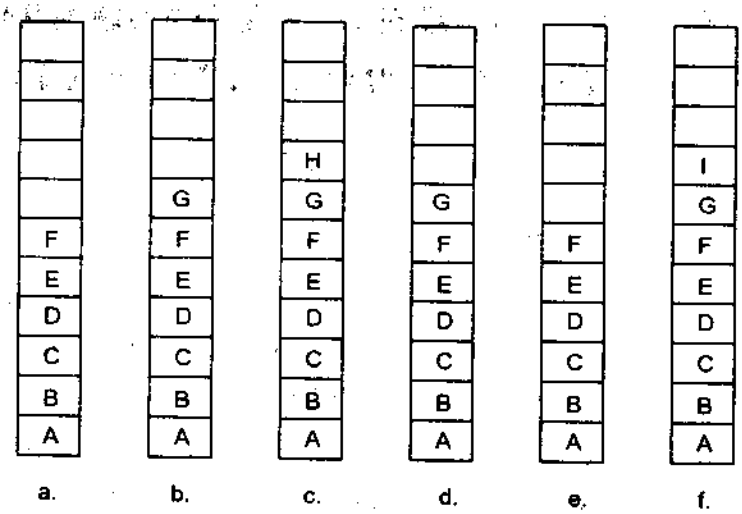
push(S, 'G') (hình 4.1b)

push(S, 'H') (hình 4.1c)

pop(S) (hình 4.1d)

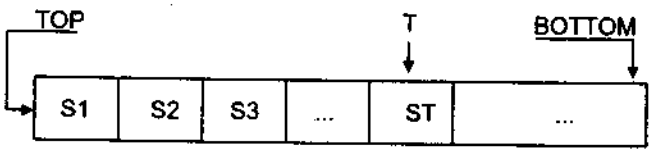
pop(S) (hình 4.1e)

push(S, 'I') (hình 4.1f)



Hình 4.1 Sự thay đổi của stack thông qua các thao tác thêm và bớt đỉnh trong stack

Có thể lưu trữ stack dưới dạng một vector S gồm n thành phần liên tiếp nhau. Nếu T là là địa chỉ của phần tử đỉnh stack thì T sẽ có giá trị biến đổi khi stack hoạt động. Ta gọi phần tử đầu tiên của stack là phần tử thứ 0, như vậy stack rỗng khi T có giá trị nhỏ hơn 0 ta qui ước là -1. Stack tràn khi T có giá trị là n -1. Mỗi khi một phần tử được thêm vào stack, giá trị của T được tăng lên 1 đơn vị, khi một phần tử bị loại bỏ khỏi stack giá trị của T sẽ giảm đi một đơn vị.



Để khai báo một stack, chúng ta có thể dùng một mảng một chiều. Phần tử thứ 0 là đáy stack, phần tử cuối của mảng là đỉnh stack. Một stack tổng quát là một cấu trúc gồm hai trường, trường top là một số

nguyên chỉ đỉnh stack. Trường node: là một mảng một chiều gồm MAX phần tử trong đó mỗi phần tử là một nút của stack. Một nút của stack có thể là một biến đơn hoặc một cấu trúc phản ánh tập thông tin về nút hiện tại. Ví dụ, khai báo stack dùng để lưu trữ các số nguyên.

```
#define TRUE  1
    #define FALSE  0
    #define MAX  100
    typedef struct {
        int top;
        int nodes[MAX];
    } stack;
```

4.1.2. Các thao tác với stack

Trong khi khai báo một stack dùng danh sách tuyến tính, chúng ta cần định nghĩa MAX đủ lớn để có thể lưu trữ được mọi đỉnh của stack. Một stack đã bị tràn (TOP = MAX - 1) thì nó không thể thêm vào phần tử trong stack, một stack rỗng thì nó không thể đưa ra phần tử. Vì vậy, chúng ta cần xây dựng thêm các thao tác kiểm tra stack có bị tràn hay không (full) và thao tác kiểm tra stack có rỗng hay không (empty).

Thao tác Empty: Kiểm tra stack có rỗng hay không:

```
int Empty(stack *ps) {
        if(ps -> top = -1)
            return(TRUE);
        return(FALSE);
}
```

Thao tác Push: Thêm nút mới x vào đỉnh stack và thay đổi đỉnh stack.

```
void Push (stack *ps, int x) {
```

```
        if(ps -> top = -1) {
            printf("\n stack full");
            return;
        }
        ps -> top = ps -> top + 1;
        ps -> nodes[ps -> top] = x;
}
```

Thao tác Pop: Loại bỏ nút tại đỉnh stack.

```
int Pop (stack *ps) {
    if(Empty(ps) {
        printf("\n stack empty");
        return(0);
    }
    return(ps -> nodes[ps -> top - -]);
}
```

4.1.3. Ứng dụng của stack

Ví dụ 4.1. Chương trình đảo ngược xâu kí tự.

Quá trình đảo ngược một xâu kí tự giống như việc đưa vào (push) từng kí tự trong xâu vào stack, sau đó đưa ra (pop) các kí tự trong stack ra cho tới khi stack rỗng ta được một xâu đảo ngược. Chương trình sau sẽ minh họa cơ chế LIFO đảo ngược xâu kí tự sử dụng stack.

```
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#include <dos.h>
#include <string.h>
#define MAX     100
#define TRUE    1
```

```
#define FALSE 0
typedef struct {
    int top;
    char node[MAX];
} stack;
/* Nguyên mẫu của hàm */
int Empty(stack *);
void Push(stack *, char);
char Pop(stack *);
/* Mô tả hàm */
int Empty(stack *ps) {
    if(ps -> top = -1)
        return(TRUE);
    return(FALSE);
}
void Push(stack *ps, char x) {
    if(ps -> top = MAX -1) {
        printf("\n Stack full");
        delay(2000);
        return;
    }
    (ps -> top) = (ps -> top) + 1;
    ps -> node[ps -> top] = x;
}
char Pop(stack *ps) {
    if(Empty(ps)) {
        printf("\n Stack empty");
        delay(2000); return(0);
```

```
    }
    return(ps -> node[ps -> top - -]);
}
void main(void) {
    stack s;
    char c, chuoi[MAX];
    int i, vitri, n; s.top = -1; clrscr();
    printf("\n Nhập string:"); gets(chuoi);
    vitri = strlen(chuoi);
    for (i = 0; i < vitri; i++)
        Push(&s, chuoi[i]);
    while(!Empty(&s))
        printf("%c", Pop(&s));
    getch();
}
```

Ví dụ 4.2. Chuyển đổi số từ hệ thập phân sang hệ cơ số bất kỳ.

Để chuyển đổi một số ở hệ thập phân thành số ở hệ cơ số bất kỳ, chúng ta lấy số đó chia cho cơ số cần chuyển đổi, lưu trữ lại phần dư của phép chia, sau đó đảo ngược lại dãy các số dư ta nhận được số cần chuyển đổi, việc làm này giống như cơ chế LIFO của stack.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define MAX 100
#define TRUE   1
#define FALSE 0
typedef struct {
    int top;
    unsigned int node[MAX];
```

```
} stack;
int Empty(stack *);
void Push(stack *, int);
int Pop(stack *);
int Empty(stack *ps) {
    if(ps -> top = -1) {
        printf("\n Stack empty");
        delay(2000); return(TRUE);
    }
    return(FALSE);
}
void Push(stack *ps, int p) {
    if(ps -> top = MAX -1) {
        printf("\n Stack full");
        delay(2000); return;
    }
    ps -> top = (ps -> top) + 1;
    ps -> node[ps -> top] = p;
}
int Pop(stack *ps) {
    if(Empty(ps)) {
        printf("\n Stack Empty");
        delay(2000); return(0);
    }
    return(ps -> node[ps -> top --]);
}
void main(void) {
    int n, coso, sodu;
```

```
    stack s; s.top = -1;
    clrscr();
    printf("\n Nhập một số n = "); scanf("%d", &n);
    printf("\n Cơ số cần chuyển:"); scanf("%d", &coso);
    while(n != 0) {
        sodu = n % coso;
        Push(&s, sodu);
        n = n/coso;
    }
    while(!Empty(&s))
        printf("%X", Pop(&s));
    getch();
}
```

Ví dụ 4.3. Tính giá trị một biểu thức dạng hậu tố.

Xét một biểu thức dạng hậu tố chỉ chứa các phép toán cộng (+), trừ (-), nhân (*), chia (/), lũy thừa ($). Cần phải nhắc lại rằng, nhà logic học Lewinski đã chứng minh được rằng, mọi biểu thức đều có thể biểu diễn dưới dạng hậu tố mà không cần dùng thêm các kí hiệu phụ. Ví dụ :

$$23+5*2\$ = ((2 + 3) *5)^2 = 625$$

Để tính giá trị của biểu thức dạng hậu tố, chúng ta sử dụng một stack lưu trữ biểu thức quá trình tính toán được thực hiện như sau:

Lấy toán hạng 1 (2) -> Lấy toán hạng 2 (3) -> Lấy phép toán '+' -> Lấy toán hạng 1 cộng toán hạng 2 và đẩy vào stack (5) -> Lấy toán hạng tiếp theo (5), lấy phép toán tiếp theo (*), nhân với toán hạng 1 rồi đẩy vào stack (25), lấy toán hạng tiếp theo (2), lấy phép toán tiếp theo ($) và thực hiện, lấy luỹ thừa rồi đẩy vào stack. Cuối cùng ta nhận được $25^2 = 625$. Chương trình tính giá trị biểu thức dạng hậu tố được thực hiện như sau:

```
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
```

```
#include <dos.h>
#include <string.h>
#include <math.h>
#define MAX 100
#define TRUE   1
#define FALSE 0
typedef struct {
    int top;
    double node[MAX];
} stack;
int Empty(stack *);
void Push(stack *, double);
double Pop(stack *);
double Dinhtri(char *);
int lakyso(char);
double tinh(int, double, double);
int Empty(stack *ps) {
    if(ps -> top = -1) {
        printf("\n Stack empty");
        delay(2000); return(TRUE);
    }
    return(FALSE);
}
void Push(stack *ps, double p) {
    if(ps -> top = MAX -1) {
        printf("\n Stack full");
        delay(2000); return;
    }
```

```
    ps -> top = (ps -> top) + 1;
    ps -> node[ps -> top] = p;
}
double Pop(stack *ps) {
    if(Empty(ps)) {
        printf("\n Stack Empty");
        delay(2000); return(0);
    }
    return(ps -> node[ps -> top - -]);
}
double Dinhtri(char *Bieuthuc) {
    int i, c, vitri;
    double toanhang1, toanhang2, giatri;
    stack s;
    s.top = -1; vitri = strlen(Bieuthuc);
    for(i = 0; i < vitri; i++) {
        if(lakyso(Bieuthuc[i]))
            Push(&s, (double)(Bieuthuc[i] -'0'));
        else {
            toanhang2 = Pop(&s);
            toanhang1 = Pop(&s);
            giatri = tinh(Bieuthuc[i], toanhang1, toanhang2);
            Push(&s, giatri);
        }
    }
    return(Pop(&s));
}
int lakyso(char kitu) {
```

```
    return(kitu >= '0' && kitu <= '9');
}
double tinh(int toantu, double toanhang1, double toanhang2) {
    double ketqua = 0;
    switch(toantu) {
        case '+': ketqua = toanhang1 + toanhang2; break;
        case ' -': ketqua = toanhang1 - toanhang2; break;
        case '*': ketqua = toanhang1 * toanhang2; break;
        case '/': ketqua = toanhang1 / toanhang2; break;
        case '$': ketqua = pow(toanhang1, toanhang2); break;
    }
    return(ketqua);
}
void main(void) {
    char c, bieuthuc[MAX];
    int vitri; clrscr();
    printf("\n Nhập một biểu thức:"); gets(bieuthuc);
    printf("\n Giá trị = %f", Dinhtri(bieuthuc));
}
```

4.2. HÀNG ĐỢI (QUEUE)

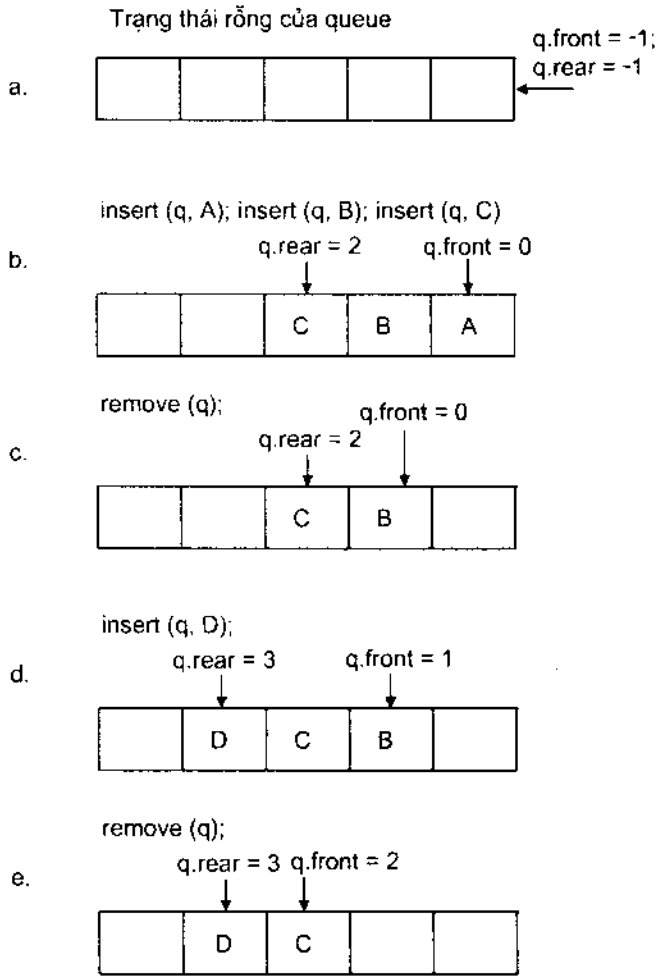
4.2.1. Giới thiệu hàng đợi

Khác với stack, hàng đợi (queue) là một danh sách tuyến tính mà thao tác bổ sung phần tử được thực hiện ở một đầu gọi là lối vào (rear). Phép loại bỏ phần tử được thực hiện ở một đầu khác gọi là lối ra (front). Như vậy, cơ chế của queue giống như một hàng đợi, đi vào ở một đầu và đi ra ở một đầu hay FIFO (First In First Out).

Để truy nhập vào hàng đợi, chúng ta sử dụng hai biến con trỏ front chỉ lối trước và rear chỉ lối sau. Khi lối trước trùng với lối sau

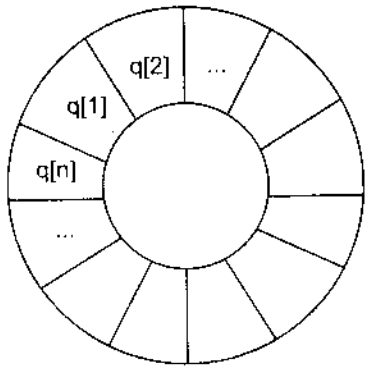
(q.front = q.rear) thì queue ở trạng thái rỗng (hình 4.2a), để thêm dữ liệu vào hàng đợi các phần tử A, B, C được thực hiện thông qua thao tác insert(q, A), insert(q, B), insert(q, C) được mô tả ở hình 4.2b, thao tác loại bỏ phần tử khỏi hàng đợi được mô tả ở hình 4.2c, những thao tác tiếp theo được mô tả tại hình 4.2d, e, f.

Cách tổ chức này sẽ dẫn tới trường hợp các phần tử di chuyển khắp không gian nhớ khi thực hiện bổ sung và loại bỏ. Ví dụ: cứ mỗi phép bổ sung kèm theo một phép loại bỏ sẽ dẫn tới trường hợp:



Hình 4.2 Các thao tác trong hàng đợi

q.front = q.rear = MAXQUE -1; và phép bổ sung và loại bỏ không thể tiếp tục thực hiện. Để khắc phục tình trạng này, chúng ta có thể tổ chức queue như một vòng tròn, khi đó q[1] coi như đứng sau q[MAXQUE -1].



Trong nhiều trường hợp, khi thực hiện thêm hoặc loại bỏ phần tử của hàng đợi chúng ta cần xét tới một thứ tự ưu tiên nào đó, khi đó hàng đợi được gọi là hàng đợi có độ ưu tiên (Priority Queue). Với priority queue, thì nút nào có độ ưu tiên cao nhất được thực hiện loại bỏ trước nhất, còn với thao tác thêm phần tử vào hàng đợi trở thành thao tác thêm phần tử vào hàng đợi có xét tới độ ưu tiên.

4.2.2. Ứng dụng hàng đợi

Mọi vấn đề của thực tế liên quan tới cơ chế FIFO như cơ chế gửi tiền, rút tiền trong ngân hàng, đặt vé máy bay... đều có thể ứng dụng được bằng hàng đợi. Hàng đợi còn có những ứng dụng trong việc giải quyết các bài toán của hệ điều hành và chương trình dịch như bài toán điều khiển các quá trình, điều khiển nạp chương trình vào bộ nhớ hay bài toán lập lịch. Sau đây là những ví dụ minh họa về ứng dụng của hàng đợi.

Ví dụ 4.4. Giải quyết bài toán "*Người sản xuất và nhà tiêu dùng*" với số các vùng đệm hạn chế.

Chúng ta hãy mô tả quá trình sản xuất và tiêu dùng như hai quá trình riêng biệt và thực hiện song hành, người sản xuất có thể sản xuất tối đa n mặt hàng,

người tiêu dùng cũng chỉ được phép sử dụng trong số n mặt hàng. Tuy nhiên, người sản xuất khi sản xuất một mặt hàng anh ta chỉ có thể lưu trữ vào kho khi và chỉ khi kho chưa bị đầy, đồng thời khi đó, nếu kho hàng không rỗng (kho có hàng) người tiêu dùng có thể tiêu dùng những mặt hàng trong kho theo nguyên tắc hàng nào nhập vào kho trước được tiêu dùng trước giống như cơ chế FIFO của queue. Sau đây là những thao tác chủ yếu trên hàng đợi để giải quyết bài toán:

Định nghĩa hàng đợi như một danh sách tuyến tính gồm MAX phần tử mỗi phần tử là một cấu trúc, hai biến front, rear trỏ lối vào và lối ra trong queue:

```
typedef   struct {
    int mahang;
    char ten[20];
} hang;
typedef struct {
    int front, rear;
    hang node[MAX];
} queue;
```

Thao tác Initialize: thiết lập trạng thái ban đầu của hàng đợi. Ở trạng thái này, font và rear có cùng một giá trị :MAX -1.

```
void Initialize (queue *pq) {
    pq -> front = pq -> rear = MAX -1;
}
```

Thao tác Empty: kiểm tra hàng đợi có ở trạng thái rỗng hay không. Hàng đợi rỗng khi front = rear.

```
int Empty(queue *pq) {
    if(pq -> front = pq -> rear)
        return(TRUE);
    return(FALSE);
}
```

Thao tác Insert: thêm X vào hàng đợi Q. Nếu việc thêm X vào hàng đợi được thực hiện ở đầu hàng thì rear có giá trị 0, nếu rear không phải ở đầu hàng đợi thì giá trị của nó được tăng lên 1 đơn vị.

```
void Insert(queue *pq, hang x) {
    if(pq -> rear = MAX -1)
        pq -> rear = 0;
    else
        (pq -> rear)++;
    if(pq -> rear = pq -> front) {
        printf("\n Queue full");
        delay(2000); return;
    }
    else
        pq -> node[pq -> rear] = x;
}
```

Thao tác Remove: loại bỏ phần tử ở vị trí front khỏi hàng đợi. Nếu hàng đợi ở trạng thái rỗng thì thao tác Remove không thể thực hiện được, trong trường hợp khác front được tăng lên một đơn vị.

```
hang  Remove(queue *pq) {
    if(Empty(pq)) {
        printf("\n Queue Empty");
        delay(2000);
    }
    else {
        if(pq -> front = MAX -1)
            pq -> front = 0;
        else
            pq -> front++;
    }
    return(pq -> node[pq -> front]);
}
```

Thao tác Traver: Duyệt tất cả các nút trong hàng đợi.

```
void Traver(queue *pq) {
    int i;
    if(Empty(pq)) {
        printf("\n Queue Empty");
        return;
    }
    if(pq -> front = MAX -1)
        i = 0;
    else
        i = pq -> front+1;
    while(i != pq -> rear) {
        printf("\n %11d % 15s", pq -> node[i].mahang, pq -> node[i].ten);
        if(i = MAX -1)
            i = 0;
        else
            i++;
    }
    printf("\n %11d % 15s", pq -> node[i].mahang, pq -> node[i].ten);
}
```

Sau đây là toàn bộ chương trình:

```
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#include <dos.h>
#include <string.h>
#include <math.h>
#define MAX 50
#define TRUE  1
```

```
#define FALSE 0
typedef struct {
    int mahang;
    char ten[20];
} hang;
typedef struct {
    int front, rear;
    hang node[MAX];
} queue;
/* Nguyên mẫu của hàm */
void   Initialize(queue *pq);
int Empty(queue *);
void Insert(queue *, hang x);
hang Remove(queue *);
void   Traver(queue *);
/* Mô tả hàm */
void Initialize (queue *pq) {
    pq -> front = pq -> rear = MAX -1;
}
int Empty(queue *pq) {
    if(pq -> front = pq -> rear)
        return(TRUE);
    return(FALSE);
}
void Insert(queue *pq, hang x) {
    if(pq -> rear = MAX -1)
        pq -> rear = 0;
    else
```

```
        (pq -> rear)++;
    if(pq -> rear = pq -> front) {
        printf("\n Queue full");
        delay(2000); return;
    }
    else
        pq -> node[pq -> rear] = x;
}
hang  Remove(queue *pq) {
    if(Empty(pq)) {
        printf("\n Queue Empty");
        delay(2000);
    }
    else {
        if(pq -> front = MAX -1)
            pq -> front = 0;
        else
            pq -> front++;
    }
    return(pq -> node[pq -> front]);
}
void Traver(queue *pq) {
    int i;
    if(Empty(pq)) {
        printf("\n Queue Empty");
        return;
    }
    if(pq -> front = MAX -1)
```

```
        i = 0;
    else
        i = pq -> front+1;
    while(i != pq -> rear) {
        printf("\n %11d % 15s", pq -> node[i].mahang, pq -> node[i].ten);
        if(i = MAX -1)
            i = 0;
        else
            i++;
    }
    printf("\n %11d % 15s", pq -> node[i].mahang, pq -> node[i].ten);
}
void main(void) {
    queue q;
    char chucnang, front1; char c; hang mh;
    clrscr();
    Initialize(&q);
    do {
        clrscr();
        printf("\n NGƯỜI SẢN XUẤT/NHÀ TIÊU DÙNG");
        printf("\n 1 - Nhập một mặt hàng");
        printf("\n 2 - Xuất một mặt hàng");
        printf("\n 3 - Xem một mặt hàng");
        printf("\n 4 - Xem hàng mới nhập");
        printf("\n 5 - Xem tất cả");
        printf("\n 6 - Xuất toàn bộ");
        printf("\n Chức năng chọn:"); chucnang = getch();
        switch(chucnang) {
```

```
        case '1':
            printf("\n Mã mặt hàng:"); scanf("%d", &mh.mahang);
            printf("\n Tên hàng:"); scanf("%s", mh.ten);
            Insert(&q, mh); break;
        case '2':
            if(!Empty(&q)) {
                mh = Remove(&q);
                printf("\n %5d %20s", mh.mahang, mh.ten);
            }
            else {
                printf("\n Queue Empty");
                delay(1000);
            }
            break;
        case '3':
            front1 = (q.front = MAX -1)?0:q.front+1;
            printf("\n Hàng xuất");
            printf("\n %6d %20s", q.node[front1].mahang, q.node[front1].ten);
            break;
        case '4':
            printf("\n Hàng mới nhập");
            printf("\n %5d %20s", q.node[q.rear].mahang, q.node[q.rear].ten);
            break;
        case '5':
            printf("\ Hàng trong kho");
            Traverse(&q); delay(2000); break;
    }
  } while(chucnang != '0');
}
```

Ví dụ 4.5. Bài toán lập lịch có ưu tiên.

Giả sử có n quá trình thực hiện song hành trong hệ thống, ở mỗi thời điểm CPU chỉ đáp ứng được cho một quá trình. Hãy lập lịch để CPU đáp ứng cho mỗi quá trình đang thực hiện trong hệ, sao cho quá trình nào có độ ưu tiên cao nhất được đáp ứng trước nhất.

Để giải quyết bài toán, chúng ta tổ chức các quá trình đang đợi CPU đáp ứng như một hàng đợi. Mỗi phần tử của hàng đợi là một quá trình và có thể được Hệ điều hành quản lý bằng một khối điều khiển các quá trình PCB (Proccess Control Block), mỗi PCB được phản ánh bằng những thông tin sau:

Pointer	Register
Memory Limited	
I/O Driver	
List Open File	
.....	
.....	
Priority:	

Để đơn giản, chúng ta coi tất cả thông tin phản ánh về quá trình như một số nguyên và số nguyên đó trùng với độ ưu tiên của quá trình. Khi đó, việc thực hiện nạp quá trình vào hàng đợi như nhập một số nguyên và nạp vào hàng đợi sao cho số lớn hơn sẽ được nạp vào phần tử đầu tiên, với cách làm như vậy dãy các quá trình sẽ tự động sắp xếp theo thứ tự giảm dần của độ ưu tiên. Quá trình CPU đáp ứng giống như việc loại bỏ quá trình khỏi hàng đợi, quá trình nào có độ ưu tiên lớn nhất sẽ được CPU đáp ứng sớm nhất. Sau đây là chương trình giải quyết bài toán lập lịch đơn giản cho CPU:

```
#include <stdio.h>

#include <stdlib.h>

#include <conio.h>

#include <dos.h>

#include <string.h>

#include <math.h>
```

```
#define MAX 100
#define TRUE   1
#define FALSE 0
typedef struct {
    int rear;
    int node[MAX];
} pqueue;
void Pqinitialize(pqueue *pq) {
    pq -> rear = -1;
}
int Pqempty(pqueue *pq) {
    if(pq -> rear = -1)
        return(TRUE);
    return(FALSE);
}
int Pqueuesize(pqueue *pq) {
    return(pq -> rear+1);
}
void Pqinsert(pqueue *pq, int x) {
    int i, j;
    if(pq -> rear = MAX - 1) {
        printf("\n Queue full");
        delay(2000); return;
    }
    for (i = 0; i < Pqueuesize(pq) && pq -> node[i]> = x; i++);
    for(j = Pqueuesize(pq); j>i; j - -)
        pq -> node[j] = pq -> node[j -1];
    pq -> node[i] = x;
```

```
    pq -> rear++;
}
int Pqremove(pqueue *pq) {
    int x, i;
    if(Pqempty(pq)) {
        printf("\n Queue full");
        delay(2000); return(x);
    }
    else {
        x = pq -> node[0];
        for (i = 0; i < pq -> rear; i++)
            pq -> node[i] = pq -> node[i+1]++;
        pq -> rear - -;
    }
    return(x);
}
void Pqtraver(pqueue *pq) {
    int i;
    if(Pqempty(pq)) {
        printf("\n Queue Empty");
        delay(2000); return;
    }
    for (i = 0; i < = pq -> rear; i++)
        printf("%5d", pq -> node[i]);
}
void main(void) {
    pqueue *pq;
    int douutien, chucnang; char c;
```

```
    Pqinitialize(pq);
    do {
        clrscr();
        printf("\n QUEUE PRIORITY");
        printf("\n 1 - Thêm nút ưu tiên");
        printf("\n 2 - Xóa nút ưu tiên");
        printf("\n 3 - Xóa hàng đợi");
        printf("\n 4 - Duyệt hàng đợi");
        printf("\n 0 - Kết thúc");
        chucnang = getch();
        switch(chucnang) {
            case '1':
                printf("\n Độ ưu tiên:"); scanf("%d", &douutien);
                Pqinsert(pq, douutien);
                break;
            case '2':
                Pqremove(pq); break;
            case '3':
                Pqinitialize(pq); break;
            case '4':Pqtraver(pq); delay(2000); break;
        }
    } while(chucnang != '0');
}
```

4.3. DANH SÁCH LIÊN KẾT ĐƠN

4.3.1. Giới thiệu và định nghĩa

Một danh sách móc nối, hoặc ngắn gọn hơn, một danh sách, là một dãy có thứ tự các phần tử được gọi là đỉnh. Danh sách có điểm bắt

đầu, gọi là tiêu đề hay đỉnh đầu, một điểm cuối cùng gọi là đỉnh cuối. Mọi đỉnh trong danh sách đều có cùng kiểu ngay cả khi kiểu này có nhiều dạng khác nhau.

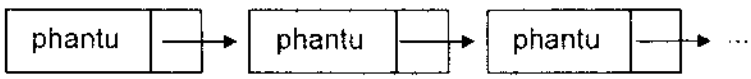
Bản chất động là một trong những tính chất chính của danh sách móc nối. Có thể thêm hoặc bớt đỉnh trong danh sách vào mọi lúc, mọi vị trí. Vì số đỉnh của danh sách không thể dự kiến trước được, nên khi thực hiện, chúng ta phải dùng con trỏ mà không dùng được mảng để bảo đảm việc thực hiện hiệu quả và tin cậy.

Mỗi đỉnh trong danh sách đều gồm hai phần. Phần thứ nhất chứa dữ liệu. Dữ liệu có thể chỉ là một biến đơn hoặc là một cấu trúc (hoặc con trỏ cấu trúc) có kiểu nào đó. Phần thứ hai của đỉnh là một con trỏ chỉ vào địa chỉ của đỉnh tiếp theo trong danh sách. Vì vậy có thể dễ dàng sử dụng các đỉnh của danh sách qua một cấu trúc tự trỏ hoặc đệ qui.

Xem như một thí dụ đơn giản, ta hãy xét trường hợp mỗi đỉnh của danh sách chỉ lưu giữ một biến nguyên. Có thể định nghĩa đỉnh như sau:

```
/* Đỉnh của danh sách đơn chỉ chứa một số nguyên */
    struct don {
        int phantu;
        struct don *tiep;
    };
    typedef struct don don_t;
```

Trong trường hợp này, biến nguyên phantu của từng đỉnh chứa dữ liệu còn biến con trỏ tiep chứa địa chỉ của đỉnh tiếp theo. Sơ đồ biểu diễn danh sách móc nối đơn được biểu diễn như hình dưới đây



Hình 4.3. Danh sách móc nối đơn

Tổng quát hơn, mỗi đỉnh của danh sách có thể chứa nhiều phần tử dữ liệu. Trong trường hợp này, hợp lý hơn cả là định nghĩa một kiểu cấu trúc tương ứng với dữ liệu cần lưu giữ tại mỗi đỉnh. Phương pháp này được sử dụng trong định nghĩa kiểu sau đây:

```
/* Đỉnh của danh sách tổng quát */
    struct tq {
        thtin_t phantu;
        struc tq *tiep;
    };
    typedef struct tq tq_t;
```

Kiểu cấu trúc *thtin_t* phải được định nghĩa trước đó để tương ứng với các dữ liệu sẽ được lưu trữ tại từng đỉnh. Danh sách được tạo nên từ kiểu đỉnh này giống như ở sơ đồ trong hình 4.3, ngoại trừ việc mỗi phantu là một biến nguyên.

4.3.2. Các thao tác trên danh sách móc nối

Thao tác các danh sách móc nối bao gồm việc cấp phát bộ nhớ cho các đỉnh (thông qua các hàm MALLOC hoặc CALLOC) và gán dữ liệu cho con trỏ. Để danh sách được tạo nên đúng đắn, ta biểu diễn cho phần tử cuối danh sách là một con trỏ NULL. Con trỏ NULL là tín hiệu thông báo không còn phần tử nào tiếp theo trong danh sách nữa.

Tiện hơn cả là chúng ta định nghĩa một con trỏ tới danh sách như sau:

```
struct node {
    int infor;
    struct node *next;
};
typedef struct node *NODEPTR;          // Con trỏ tới node
```

Cấp phát bộ nhớ cho một node:

```
NODEPTR Getnode(void) {
        NODEPTR p;
        P = (NODEPTR) malloc(sizeof(struct node));
        return(p);
}
```

Giải phóng bộ nhớ của một node:

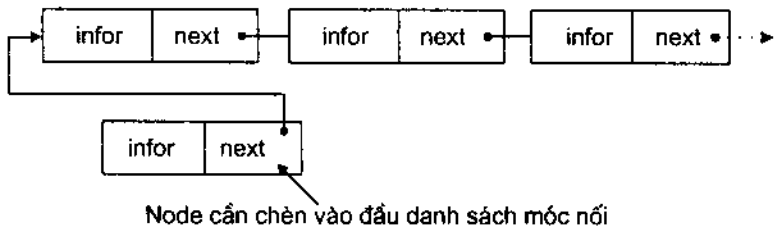
```
NODEPTR Freenode(NODEPTR p) {
        free(p);
    }
```

Chèn một phần tử mới vào đầu danh sách:

Các bước để chèn một phần tử mới vào đầu danh sách cần thực hiện là:

- Cấp không gian bộ nhớ đủ lưu giữ một đỉnh mới;
- Gán các giá trị con trỏ thích hợp cho đỉnh mới;
- Thiết lập liên kết với đỉnh mới.

Sơ đồ biểu diễn phép thêm một đỉnh mới vào đầu danh sách được thể hiện như hình 4.4.



Hình 4.4. Thêm đỉnh mới vào đầu danh sách móc nối đơn

```
void Push-Top(NODEPTR *plist, int x) {
        NODEPTR p;
```

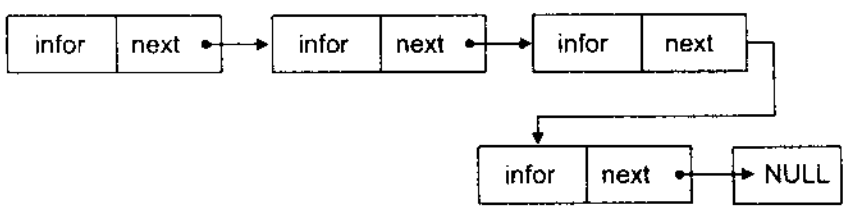
```
        p = Getnode();          // cấp không gian nhớ cho đỉnh mới
        p -> infor = x;         // gán giá trị thích hợp cho đỉnh mới
        p -> next = *plist;
        *plist = p;             // thiết lập liên kết
}
```

Thêm một phần tử mới vào cuối danh sách:

Để thêm một node vào cuối danh sách, ta cần thực hiện qua các bước sau:

- Cấp phát bộ nhớ cho node mới;
- Gán giá trị thích hợp cho node mới;
- Di chuyển con trỏ tới phần tử cuối danh sách;
- Thiết lập liên kết cho node mới.

Sơ đồ thể hiện phép thêm một phần tử mới vào cuối danh sách được thể hiện như trong hình 4.5.



Hình 4.5. Thêm node mới vào cuối danh sách.

```
void Push_Bottom(NODEPTR *plist, int x) {
        NODEPTR p, q;
        p = Getnode();          // cấp phát bộ nhớ cho node mới
        p -> infor = x;         // gán giá trị thông tin thích hợp
        q = *plist;             // chuyển con trỏ tới cuối danh sách
        while(q -> next != NULL)
        q = q -> next;
```

```
        // q là node cuối cùng của danh sách liên kết
        q -> next = p;          //node cuối bây giờ là node p;
        p -> next = NULL;       // liên kết mới của p
    }
```

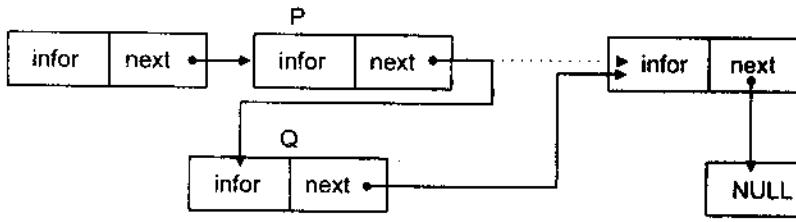
Thêm node mới vào giữa danh sách (trước node p):

Để thêm node q vào trước node p, chúng ta cần lưu ý node p phải có thực trong danh sách. Giả sử node p là có thực, khi đó xảy ra hai tình huống: hoặc node p là node cuối cùng của danh sách liên kết tức p -> next = NULL, hoặc node p chưa phải là cuối cùng hay p -> next != NULL. Trường hợp thứ nhất, chúng ta chỉ cần gọi tới thao tác Push_Bottom(). Trường hợp thứ 2, chúng ta thực hiện theo các bước như sau:

- Cấp phát bộ nhớ cho node mới;
- Gán giá trị thích hợp cho node;
- Thiết lập liên kết node q với node kế tiếp p;
- Thiết lập liên kết node node p với node q;

```
void Push_Before(NODEPTR p, int x) {
        NODEPTR q;
        if(p -> next = NULL)
                Push_Bottom(p, x);
        else {
                q = Getnode();          // cấp phát bộ nhớ cho node mới
                q -> infor = x;         // gán giá trị thông tin thích hợp
                q -> next = p -> next;
                        // thiết lập liên kết node q với node kế tiếp p;
                p -> next = q;
                        // thiết lập liên kết node p với node kế tiếp q;
        }
}
```

Sơ đồ thêm node vào giữa danh sách được thể hiện như sau:



Hình 4.6. Phép thêm phần tử vào giữa danh sách liên kết đơn

Xoá một node ra khỏi đầu danh sách:

Khi loại bỏ node khỏi đầu danh sách liên kết, chúng ta cần chú ý rằng nếu danh sách đang rỗng thì không cần phải loại bỏ. Trong trường hợp còn lại, ta thực hiện như sau:

- Dùng node p trỏ tới đầu danh sách;
- Dịch chuyển vị trí đầu danh sách tới node tiếp theo;
- Loại bỏ liên kết với p;
- Giải phóng node p;

```
void Del_Top(NODEPTR *plist) {
        NODEPTR p;
        p = *plist;                 // node p trỏ tới đầu danh sách;
        if(p = NULL) return;        // danh sách rỗng
        (*plist) = (*plist) -> next;
                                    // dịch chuyển node gốc lên node kế tiếp
        p -> next = NULL;           // loại bỏ liên kết với p
        Freenode(p);                // giải phóng p;
}
```

Loại bỏ node ở cuối danh sách:

Một node ở cuối danh sách có thể xảy ra ba tình huống sau:

- Danh sách rỗng: ta không cần thực hiện loại bỏ;
- Danh sách chỉ có đúng một node: ứng với trường hợp loại bỏ node gốc;
- Trường hợp còn lại danh sách có nhiều hơn một node, khi đó ta phải dịch chuyển tới node gần node cuối cùng nhất để thực hiện loại bỏ.

```
void Del_Bottom(NODEPTR *plist) {
        NODEPTR p, q;
        if(*plist = NULL) return;              // không làm gì
        else if((*plist) -> next = NULL))      // danh sách có một node
                Del_Top(plist);
            else {
                p = *plist;
                while(p -> next != NULL) {
                        q = p;
                        p = p -> next;         // q là node sau node p;
            }
          // p là node cuối danh sách;
          q -> next = NULL;                    // node cuối cùng là q
          Freenode(p);                         // giải phóng p;
        }
}
```

Loại bỏ node ở giữa danh sách (trước node p):

Cần để ý rằng, nếu trước node p là NULL (p -> next = NULL) thì ta không thực hiện loại bỏ được. Trường hợp còn lại chúng ta thực hiện như sau:

- Dùng node q trỏ tới node trước node p;

- Loại bỏ liên kết của q;
- Giải phóng q.

```
void Del_before(NODEPTR p) {
        NODEPTR q;
        if(p -> next = NULL) return;            // không làm gì
        q = p -> next;
        p -> next = q -> next;
        Freenode(q);
}
```

4.3.4. Ứng dụng của danh sách liên kết đơn

Ví dụ viết chương trình quản lý sinh viên sau sẽ minh hoạ đầy đủ cho các thao tác trên danh sách đơn.

Ví dụ 4.6. Viết chương trình quản lý sinh viên bằng danh sách móc nối đơn.

Để đơn giản, chúng ta chỉ quản lý hai thuộc tính mã sinh viên (masv) và họ tên sinh viên (hoten), còn việc mở rộng bài toán coi như một bài tập thực hành.

```
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#include <dos.h>
#include <string.h>
#include <math.h>
#include <alloc.h>
#define TRUE   1
#define FALSE 0
typedef struct {
   int masv;
   char hoten[20];
```

```
} sinhvien;
typedef struct node {
    sinhvien infor;
    struct node *next;
} *NODEPTR;
void Initialize(NODEPTR *plist) {
    *plist = NULL;
}
NODEPTR Getnode(void) {
    NODEPTR p;
    p = (NODEPTR) malloc(sizeof(struct node));
    return(p);
}
void Freenode(NODEPTR p) {
    free(p);
}
int Emptynode(NODEPTR *plist) {
    if(*plist = NULL)
        return(TRUE);
    return(FALSE);
}
NODEPTR Inserttop(NODEPTR *plist, sinhvien x) {
    NODEPTR p;
    p = Getnode();
    p -> infor = x;
    if(Emptynode(plist)) {
        p -> next = NULL;
        *plist = p;
```

```
        return(p);
    }
    p -> next = *plist;
    *plist = p;
    return(p);
}
int Bottomnode(NODEPTR *plist) {
    int i; NODEPTR p;
    if(Emptynode(plist))
        return(-1);
    p = *plist; i = 0;
    while(p != NULL) {
        i = i+1;
        p = p -> next;
    }
    return(i);
}
NODEPTR Insertbottom(NODEPTR *plist, sinhvien x) {
    NODEPTR p, q; int n, i;
    n = Bottomnode(plist);
    if(n = -1) {
        Insertop(plist, x);
        return(*plist);
    }
    p = *plist; i = 0; q = Getnode(); q -> infor = x;
    while(i < n -1) {
        p = p -> next;
        i = i+1;
```

```
    }
    p -> next = q; q -> next = NULL;
    delay(2000); return(q);
}
NODEPTR insertafter(NODEPTR *plist, sinhvien x, int n) {
    NODEPTR p, q; int i;
    if(n < 0) {
        printf("\n Vị trí không hợp lệ");
        delay(2000); return(NULL);
    }
    p = *plist; i = 0;
    while(p != NULL && i < n) {
        i = i+1;
        p = p -> next;
    }
    if(p = NULL) {
        printf("\n Vị trí không hợp lệ");
        delay(2000); return(NULL);
    }
    q = Getnode(); q -> infor = x;
    q -> next = p -> next;
    p -> next = q;
    return(q);
}
void Deltop(NODEPTR *plist) {
    NODEPTR p, q;
    p = *plist;
    if(Emptynode(plist)) {
```

```
        printf("\n Danh sách rỗng");
        delay(2000); return;
    }
    q = p; p = p -> next; *plist = p;
    printf("\n Node bị loại bỏ");
    printf("\n% -5d% -20s", q -> infor.masv, q -> infor.hoten);
    delay(2000); Freenode(q);
}
void Delbottom(NODEPTR *plist) {
    NODEPTR p, q; int i, n;
    n = Bottomnode(plist);
    if(n = -1) {
        printf("\n Danh sách rỗng");
        delay(2000); return;
    }
    if(n = 1) {
        Deltop(plist); return;
    }
    p = *plist; i = 0;
    while(i < n -2) {
        p = p -> next;
        i = i+1;
    }
    q = p -> next; p -> next = NULL;
    printf("\n Node được loại bỏ");
    printf("\n % -5d% -20s", q -> infor.masv, q -> infor.hoten);
    delay(2000); Freenode(q);
}
```

```
void Delcurrent(NODEPTR *plist, int n) {
    NODEPTR p, q; int i;
    if(Emptynode(plist)) {
        printf("\n Danh sách rỗng");
        delay(2000); return;
    }
    if(n = 0) {
        Deltop(plist); return;
    }
    p = *plist; i = 0;
    while(p != NULL && i < n -1) {
        i = i+1;
        p = p -> next;
    }
    if(p -> next = NULL) {
        printf("\n Node không hợp lệ");
        delay(2000); return;
    }
    q = p -> next; p -> next = q -> next;
    printf("\n Node được loại bỏ");
    printf("\n % -5d% -20s", q -> infor.masv, q -> infor.hoten);
    delay(2000); Freenode(q);
}
void Travenode(NODEPTR *plist) {
    NODEPTR p;
    if(Emptynode(plist)) {
        printf("\n Danh sách rỗng");
        delay(2000); return;
```

```
    }
    p = *plist;
    while(p != NULL) {
        printf("\n % -5d% -20s", p -> infor.masv, p -> infor.hoten);
        p = p -> next;
    }
    delay(2000);
}
void Sortnode(NODEPTR *plist) {
    NODEPTR p, q; sinhvien temp;
    for(p = *plist; p != NULL; p = p -> next) {
        for(q = p -> next; q != NULL; q = q -> next) {
            if(p -> infor.masv>q -> infor.masv) {
                temp = p -> infor; p -> infor = q -> infor;
                q -> infor = temp;
            }
        }
    }
    printf("\n Danh sách được sắp xếp");
    for(p = *plist; p != NULL; p = p -> next) {
        printf("\n % -5d% -20s", p -> infor.masv, p -> infor.hoten);
    }
    delay(2000);
}
void Searchnode(NODEPTR *plist, int masv) {
    NODEPTR p;
    p = *plist;
    while(p != NULL && p -> infor.masv != masv)
```

```
        p = p -> next;
    if(p = NULL)
        printf("\n Node không tồn tại");
    else {
        printf("\n Node cần tìm");
        printf("\n % -5d% -20s", p -> infor.masv, p -> infor.hoten);
    }
    delay(2000);
}

void Thuchien(void) {
    NODEPTR plist; sinhvien x, y; int vitri; char c;
    Initialize(&plist);
    do {
        clrscr();
        printf("\n THAO TÁC VỚI SINGLE LINK LIST");
        printf("\n 1 - Thêm node đầu danh sách");
        printf("\n 2 - Thêm node cuối dach sách");
        printf("\n 3 - Thêm node giữa danh sách");
        printf("\n 4 - Loại bỏ node đầu danh sách");
        printf("\n 5 - Loại bỏ node cuối danh sách");
        printf("\n 6 - Loại bỏ node giữa danh sách");
        printf("\n 7 - Duyệt danh sách");
        printf("\n 8 - Sắp xếp danh sách");
        printf("\n 9 - Tìm kiếm danh sách");
        printf("\n 0 - Trở về");
        c = getch();
```

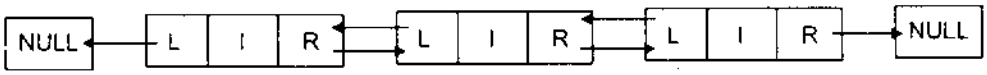
```
    switch(c) {
        case '1':
            printf("\n Mã sinh viện:"); scanf("%d", &x.masv);
            fflush(stdin); printf("\n Họ và tên:"); gets(x.hoten);
            Inserttop(&plist, x); break;
        case '2':
            printf("\n Mã sinh viênn:"); scanf("%d", &x.masv);
            fflush(stdin); printf("\n Họ và tên:"); gets(x.hoten);
            Insertbottom(&plist, x); break;
        case '3':
            printf("\n Vị trí trên:"); scanf("%d", &vitri);
            printf("\n Mã sinh viên:"); scanf("%d", &x.masv);
            fflush(stdin); printf("\n Họ và tên:"); gets(x.hoten);
            Insertafter(&plist, x, vitri - 1); break;
        case '4': Deltop(&plist); break;
        case '5': Delbottom(&plist); break;
        case '6':
            fflush(stdin); printf("\n Vị trí loại bỏ:");
            scanf("%d", &vitri);
            Delcurrent(&plist, vitri - 1); break;
        case '7': Travenode(&plist); break;
        case '8': Sortnode(&plist); break;
        case '9':
            fflush(stdin); printf("\n Mã sinh viên:");
            scanf("%d", &vitri);
            Searchnode(&plist, vitri); break;
    }
} while(c != '0');
```

```
}
void main(void) {
    Thuchien();
}
```

4.4. DANH SÁCH LIÊN KẾT KÉP

Mỗi khi thao tác trên danh sách, việc duyệt danh sách theo cả hai chiều tỏ ra thuận tiện hơn cho người sử dụng. Đôi khi chúng ta phải di chuyển trong danh sách từ node cuối lên node đầu hoặc ngược lại bằng cách đi qua một loạt các con trỏ. Điều này có thể dễ dàng giải quyết được nếu ta tăng thông tin chứa tại từng đỉnh của danh sách. Ngoài con trỏ chứa địa chỉ đỉnh tiếp theo, ta thêm con trỏ trước để chứa địa chỉ đứng sau đỉnh này. Làm như vậy, chúng ta thu được một cấu trúc dữ liệu mới gọi là danh sách liên kết kép.

```
struct node {
        int infor;
        struct node *right;          // con trỏ tới node sau
        struct node *left;           // con trỏ tới node kế tiếp
};
typedef struct node *NODEPTR;        // định nghĩa con trỏ tới node
```



Hình 4.7. Mô tả một danh sách liên kết kép.

Các thao tác trên danh sách liên kết kép cũng tương tự như danh sách liên kết đơn. Nhưng cần chú ý rằng, mỗi node p của danh sách liên kết kép có hai đường liên kết là p -> left và p -> right;

Thao tác thêm node mới vào đầu danh sách liên kết kép:

- Cấp phát bộ nhớ cho node mới;

- Gán giá trị thích hợp cho node mới;
- Thiết lập liên kết cho node mới;

```
void Push_Top(NODEPTR *plist, int x) {
        NODEPTR p;
        p = Getnode();              // cấp phát bộ nhớ cho node
        p -> infor = x;             // gán giá trị thích hợp;
        p -> right = *plist;        // thiết lập liên kết phải
        (*plist) -> left = p;       // thiết liên kết với *plist
        p -> left = NULL;           // thiết lập liên kết trái
        *plist = p;
}
```

Thao tác thêm node vào cuối danh sách:

- Nếu danh sách rỗng thì thao tác này trùng với thao tác thêm node mới vào đầu danh sách.
- Nếu danh sách không rỗng chúng ta thực hiện như sau:
- Cấp phát bộ nhớ cho node;
- Gán giá trị thích hợp cho node;
- Chuyển con trỏ tới node cuối trong danh sách;
- Thiết lập liên kết trái;
- Thiết lập liên kết phải;

```
void Push_Bottom(NODEPTR *plist, int x) {
        NODEPTR p, q;
        if(*plist = NULL)
            Push_Top(plist, x);
        else {
```

```
            p = Getnode();          // cấp phát bộ nhớ cho node
            p -> infor = x;         // gán giá trị thích hợp
            q = *plist;             // chuyển con trỏ tới node cuối danh sách
            while(q -> right != NULL)
            q = q -> right;         // q là node cuối cùng trong danh sách
            q -> right = p;         // liên kết phải
            p -> left = q;          // liên kết trái
            p -> right = NULL;      // liên kết phải
        }
    }
```

Thêm node vào trước node p:

Muốn thêm node vào trước node p thì node p phải tồn tại trong danh sách. Nếu node p tồn tại thì có thể xảy ra hai trường hợp: hoặc node p là node cuối cùng của danh sách hoặc node p là node chưa phải là cuối cùng. Trường hợp thứ nhất ứng với thao tác Push_Bottom. Trường hợp thứ hai, chúng ta làm như sau:

- Cấp phát bộ nhớ cho node;
- Gán giá trị thích hợp;
- Thiết lập liên kết trái cho node mới;
- Thiết lập liên kết phải cho node mới;

Quá trình được mô tả bởi thủ tục sau:

```
void Push_Before(NODEPTR p, int x) {
    NODEPTR q;
    if(p = NULL) return;                    // không làm gì
    else   if(p -> next = NULL)
               Push_Bottom(p, x);
           else {
```

```
            q = Getnode();              // cấp phát bộ nhớ cho node mới
            q -> infor = x;             // gán giá trị thông tin thích hợp
            q -> right = p -> right;    // thiết lập liên kết phải
            (p -> right) -> left = q;
            q -> left = p;              // thiết lập liên kết trái
            p -> right = q;
        }
}
```

Loại bỏ node đầu danh sách:

- Nếu danh sách rỗng thì không cần loại bỏ;
- Dùng node p trỏ tới đầu danh sách;
- Chuyển gốc lên node kế tiếp;
- Loại bỏ liên kết với node p;
- Giải phóng p;

```
void Del_Top(NODEPTR *plist) {
    NODEPTR p;
    if((*plist) = NULL) return;     // không làm gì
    p = *plist;                     // p là node đầu tiên trong danh sách
    (*plist) = (*plist) -> right;   // chuyển node gốc tới node kế tiếp
    p -> right = NULL;              // ngắt liên kết phải của p;
    (*plist) -> left = NULL;        // ngắt liên kết trái với p
    Freenode(p);                    // giải phóng p
}
```

Loại bỏ node ở cuối danh sách:

- Nếu danh sách rỗng thì không cần loại bỏ;
- Nếu danh sách có một node thì nó là trường hợp loại phần tử ở đầu danh sách;

- Nếu danh sách có nhiều hơn một node thì
- Chuyển con trỏ tới node cuối cùng;
- Ngắt liên kết trái của node;
- Ngắt liên kết phải của node;
- Giải phóng node.

```
void Del_Bottom(NODEPTR *plist) {
        NODEPTR p, q;
        if((*plist) = NULL) return;                    // không làm gì
        else if((*plist) -> right = NULL) Del_Top(plist);
        else {
                p = *plist;     // chuyển con trỏ tới node cuối danh sách
                while(p -> right != NULL)
                        p = p -> right;
                // p là node cuối của danh sách
                q = p -> left;                  // q là node sau p;
                q -> right = NULL;              // ngắt liên kết phải của q
                p -> left = NULL;               // ngắt liên kết trái của p
        Freenode(p);                            // giải phóng p
        }
}
```

Loại node trước node p:

- Nếu node p không có thực thì không thể loại bỏ;
- Nếu node p là node cuối thì cũng không thể loại bỏ;
- Trường hợp còn lại được thực hiện như sau:
  - Ngắt liên kết trái với node p đồng thời thiết lập liên kết phải với node (p -> right) -> right;

- Ngắt liên kết phải với node p đồng thời thiết lập liên kết trái với node (p -> right) -> right;
- Giải phóng node p -> right.

```
void Del_Before(NODEPTR p) {
        NODEPTR q, r;
        if(p = NULL || p -> right = NULL) return;
/* Không làm gì nếu node p là không có thực hoặc là node cuối cùng */
        q = (p -> right) -> right;   // q là node trước node p -> right
        r = p -> right;              // r là node cần loại bỏ
        r -> left = NULL;            // ngắt liên kết trái của r
        r -> right = NULL;           // ngắt liên kết phải của r
        p -> right = q;              // thiết lập liên kết phải mới cho p
        q -> left = p;               // thiết lập liên kết trái mới cho p
        Freenode(r);                 // giải phóng node
}
```

Chúng ta có thể xây dựng thêm các thao tác loại bỏ node bên trái, duyệt trái, duyệt phải trên danh sách móc nối kép. Những thao tác đó được thể hiện thông qua ví dụ sau.

Ví dụ 4.7. Cung cấp thông tin về tuyến xe lửa. Bao gồm: thông tin về mỗi hành trình, các hành trình đi xuôi, các hành trình đi ngược của mỗi đoàn tàu.

```
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#include <dos.h>
#include <string.h>
#include <math.h>
#include <alloc.h>
```

```
#define TRUE 1
#define FALSE 0
/* Cấu trúc thông tin chung về đoàn tàu */
typedef struct {
    char gatruoc[20];
    char gasau[20];
    int chieudai;
    int thoigian;
} doan;
/* Cấu trúc của một nút trong danh sách liên kết kép */
typedef struct node {
    doan infor;
    struct node *left, *right;
};
typedef struct node *NODEPTR;
/* Cấp phát một nút cho danh sách liên kết kép */
NODEPTR Getnode(void) {
    NODEPTR p;
    p = (NODEPTR) malloc(sizeof(struct node));
    return(p);
}
/* Giải phóng một nút của danh sách liên kết kép */
void Freenode(NODEPTR p) {
    free(p);
}
/* Khởi động danh sách liên kết kép */
void Initialize(NODEPTR *plist) {
    *plist = NULL;
}
```

```
/* Kiểm tra tính rỗng của danh sách liên kết kép */
int Empty(NODEPTR *plist) {
    if(*plist = NULL)
        return(TRUE);
    return(FALSE);
}
/* Xác định số nút trong danh sách liên kết kép */
int Listsize(NODEPTR *plist) {
    NODEPTR p; int n;
    p = *plist; n = 0;
    while(p != NULL) {
        p = p -> right;
        n++;
    }
    return(n);
}
/* Xác định con trỏ chỉ nút thứ i trong danh sách liên kết kép */
NODEPTR Nodepointer(NODEPTR *plist, int i) {
    NODEPTR p; int vitri;
    p = *plist; vitri = 0;
    while(p != NULL && vitri < i) {
        p = p -> right;
        vitri++;
    }
    return(p);
}
/* Xác định vị trí của nút p trong danh sách liên kết kép */
int Position(NODEPTR *plist, NODEPTR p) {
    int vitri; NODEPTR q;
```

```
    q = *plist; vitri = 0;
    while(q != NULL && q != p) {
        q = q -> right;
        vitri++;
    }
    if(q = NULL)
        return(-1);
    return(vitri);
}
/* Thêm nút mới vào đầu danh sách liên kết kép */
void Push(NODEPTR *plist, doan x) {
    NODEPTR p; p = Getnode();
    p -> infor = x;
    if(*plist = NULL) {
        p -> left = NULL;
        p -> right = NULL;
        *plist = p;
    }
    else {
        p -> right = *plist;
        (*plist) -> left = p;
        p -> left = NULL;
        *plist = p;
    }
}
/* Thêm nút mới vào sau nút p */
void Insertright(NODEPTR p, doan x) {
```

```
    NODEPTR q, r;
    if(p = NULL) {
        printf("\n Node không có thực");
        delay(2000); return;
    }
    else {
        q = Getnode();
        q -> infor = x;
        r = p -> right;
        r -> left = q;
        q -> right = r;
        q -> left = p;
        p -> right = q;
    }
}
/* Thêm nút mới vào trước nút p */
void Insertleft(NODEPTR *plist, NODEPTR p, doan x) {
    NODEPTR q, r;
    if(p = NULL) {
        printf("\n Node không có thực");
        delay(2000); return;
    }
    if(p = *plist)
        Push(plist, x);
    else {
        q = Getnode();
        q -> infor = x;
        r = p -> left;
```

```
        r -> right = q;
        q -> left = r;
        q -> right = p;
        p -> left = q;
    }
}
/* Xoá nút ở đầu danh sách */
doan Pop(NODEPTR *plist) {
    NODEPTR p; doan x;
    if(Empty(plist)) {
        printf("\n Danh sách rỗng");
        delay(2000);
    }
    else {
        p = *plist;
        x = p -> infor;
        if((*plist) -> right = NULL)
            *plist = NULL;
        else {
            *plist = p -> right;
            (*plist) -> left = NULL;
        }
        Freenode(p);
    }
    return(x);
}
/* Xoá nút có con trỏ là p trong danh sách */
```

```
doan Delnode(NODEPTR *plist, NODEPTR p) {
    NODEPTR q, r; doan x;
    if(p = NULL) {
        printf("\n Node không có thực");
        delay(2000); return(x);
    }
    if(*plist = NULL) {
        printf("\n Danh sách rỗng");
        delay(2000);
    }
    else {
        x = p -> infor;
        q = p -> left;
        r = p -> right;
        r -> left = q;
        q -> right = r;
        Freenode(p);
    }
    return(x);
}
/* Duyệt danh sách từ trái sang phải */
void Righttraverse(NODEPTR *plist) {
    NODEPTR p; int stt;
    if(Empty(plist)) {
        printf("\n Không có đoàn nào");
        delay(2000); return;
    }
    p = *plist;     stt = 0;
```

```
    while(p != NULL) {
        printf ("\n %5d %20s %20s %7d%7d", stt++, p -> infor.gatruoc,
            p -> infor.gasau, p -> infor.chieudai, p -> infor.thoigian);
            p = p -> right;
    }
}
/* Duyệt danh sách từ phải sang trái */
void Lefttraverse(NODEPTR *plist) {
    NODEPTR p; int stt;
    if(Empty(plist)) {
        printf("\n Không có đoàn nào");
        delay(2000); return;
    }
    stt = 0; p = Nodepointer(plist, Listsize(plist) -1);
    while(p != NULL) {
        printf("\n %5d %20s%20s%7d%7d", stt++, p -> infor.gasau,
            p -> infor.gatruoc, p -> infor.chieudai, p -> infor.thoigian);
        p = p -> left;
    }
}
/* Tìm thông tin về ga trước */
NODEPTR Search1(NODEPTR *plist, char x[]) {
    NODEPTR p = *plist;
    while(strcmp(p -> infor.gatruoc, x) != 0 && p != NULL)
        p = p -> right;
    return(p);
}
```

```
/* Tìm thông tin về ga sau */
NODEPTR Search2(NODEPTR *plist, char x[]) {
    NODEPTR p = *plist;
    while(strcmp(p -> infor.gasau, x) != 0 && p != NULL)
        p = p -> right;
    return(p);
}
/* Loại bỏ toàn bộ các nút của danh sách */
void Clearlist(NODEPTR *plist) {
    while(*plist != NULL) {
        Pop(plist);
    }
}
/* Báo lộ trình các tuyến */
void Message(NODEPTR *plist, char noidi[], char noiden[], char c) {
    NODEPTR p, p1; int kc, tg;
    if(c = 'x') {
        p = Search1(plist, noidi);
        if(p = NULL) {
            printf("\n Không có lộ trình");
            delay(2000); return;
        }
        if(strcmp(noidi, noiden) = 0) {
                printf("\n Nơi đi trùng nơi đến");
                delay(2000); return;
        }
        p1 = Search2(plist, noiden);
```

```
    if(p1 = NULL) {
        printf("\n Nơi đến không có thực");
        delay(2000); return;
    }
    if(Position(plist, p) < = Position(plist, p1)) {
        kc = tg = 0;
        while(p != p1) {
            kc = kc + p -> infor.chieudai;
            tg = tg + p -> infor.thoigian;
            printf("\n %20s -> %20s: %7d km %7d giờ",
                    p -> infor.gatruoc,
                    p -> infor.gasau, p -> infor.chieudai,
                    p -> infor.thoigian);
            p = p -> right;
        }
        kc = kc + p1 -> infor.chieudai;
        tg = tg + p1 -> infor.thoigian;
        printf("\n %20s -> %20s: %7d km %7d giờ",
                p1 -> infor.gatruoc,
                p1 -> infor.gasau, p1 -> infor.chieudai, p1 -> infor.thoigian);
                printf("\n Tổng chiều dài:% 7d Thời gian:%7d", kc, tg);
        delay(2000);
    }
}
else {
    printf("\n Không đi xuôi được");
    delay(2000); return;
}
```

```
if(c = 'n') {
    p = Search2(plist, noidi);
    if(p = NULL) {
        printf("\n Không có lộ trình");
        delay(2000); return;
    }
    if(strcmp(noidi, noiden) = 0) {
            printf("\n Nơi đi trùng nơi đến");
            delay(2000); return;
    }
    p1 = Search1(plist, noiden);
    if(p1 = NULL) {
        printf("\n Nơi đến không có thực");
        delay(2000); return;
    }
    if(Position(plist, p) < = Position(plist, p1)) {
        kc = tg = 0;
        while(p != p1) {
            kc = kc + p -> infor.chieudai;
            tg = tg + p -> infor.thoigian;
            printf("\n %20s -> %20s: %7d km %7d giờ",
                    p -> infor.gatruoc,
                    p -> infor.gasau, p -> infor.chieudai,
                    p -> infor.thoigian);
                    p = p -> right;
        }
        kc = kc + p1 -> infor.chieudai;
        tg = tg + p1 -> infor.thoigian;
```

```
            printf("\n %20s -> %20s: %7d km %7d giờ",
p1 -> infor.gatruoc,
            p1 -> infor.gasau, p1 -> infor.chieudai, p1 -> infor.thoigian);
            printf("\n Tổng chiều dài:% 7d Thoi gian:%7d", kc, tg);
            delay(2000);
        }
    }
    else {
        printf("\n Không đi ngược được");
        delay(2000); return;
    }
}
void main (void) {
    NODEPTR plist, p, p1; doan ga; char c, noidi[20], noiden[20];
    int vitri, chucnang;
    Initialize(&plist);
    do {
        clrscr();
        printf("\n QUAN SÁT TRÊN TÀU");
        printf("\n 1 - Thêm một đoàn");
        printf("\n 2 - Loại bỏ một đoàn");
        printf("\n 3 - Xem lộ trình 1");
        printf("\n 4 - Xem lộ trình 2");
        printf("\n 5 - Xem thông tin đoàn i");
        printf("\n 6 - Hiệu chỉnh thông tin đoàn i");
        printf("\n 7 - Báo lộ trình");
        printf("\n 0 - Kết thúc chương trình");
        printf("\n Lựa chọn chức năng:"); scanf("%d", &chucnang);
```

```
switch(chucnang) {
    case 1:
        printf("\n Vị trí cần thêm:"); scanf("%d", &vitri);
        printf("\n Tên ga trước:"); scanf("%s", ga.gatruoc);
        printf("\n Tên ga sau:"); scanf("%s", ga.gasau);
        printf("\n Chiều dài:"); scanf("%d", &ga.chieudai);
        printf("\n Thời gian:"); scanf("%d", &ga.thoigian);
        if(vitri = 0)
            Push(&plist, ga);
        else
            Insertright(Nodepointer(&plist, vitri -1), ga);
        break;
    case 2:
        printf("\n Vị trí:"); scanf("%d", &vitri);
        p = Nodepointer(&plist, vitri);
        if(p = NULL) {
            printf("\n Vị trí không hợp lệ");
        }
        else {
            if(vitri = 0) Pop(&plist);
            else Delnode(&plist, p);
        }
        delay(2000); break;
    case 3:
        printf("\n LỘ TRÌNH DUYỆT XUÔI");
        Righttraverse(&plist); delay(2000); break;
    case 4:
        printf("\n LỘ TRÌNH DUYỆT NGƯỢC");
        Lefttraverse(&plist); delay(2000); break;
```

```
case 5:
    printf("\n Vị trí:"); scanf("%d", &vitri);
    p = Nodepointer(&plist, vitri);
    if(p = NULL)
        printf("\n Vị trí không hợp lệ");
    else {
            printf("\n Đoàn:%d Từ:%s Đến:%s Chiều dài:%d Thời
            gian:%d", vitri, p -> infor.gatruoc, p -> infor.gasau,
                    p -> infor.chieudai,
            p -> infor.thoigian);
    }
    delay(2000); break;
case 6:
    printf("\n Vị trí:"); scanf("%d", &vitri);
    p = Nodepointer(&plist, vitri);
    if(p = NULL)
        printf("\n Vị trí không hợp lệ");
    else {
            printf("\n Đoàn:%d Từ:%s Đến:%s Chiều dài:%d Thời gian
                    :%d", vitri, p -> infor.gatruoc, p -> infor.gasau,
                        p -> infor.chieudai, p -> infor.thoigian);
            printf("\n Tên ga trước:%s"); scanf("%s",  ga.gatruoc);
            printf("\n Tên ga sau:%s"); scanf("%s", ga.gasau);
            printf("\n Chiều dài:%d"); scanf("%d", &ga.chieudai);
            printf("\n Thời gian:%d"); scanf("%d", &ga.thoigian);
    }
    delay(2000); break;
case 7:
```

```
            printf("\n Đi xuôi:x Đi ngược: n:"); c = getche();
            printf("\n Nơi đi:"); scanf("%s", noidi);
            printf("\n Nơi đến:"); scanf("%s", noiden);
            Message(&plist, noidi, noiden, c);
            delay(2000); break;
        }
    } while(chucnang != 0);
}
```

BÀI TẬP CHƯƠNG 4

4.1. Xâu thuận nghịch độc là xâu bit nhị phân có độ dài n mà khi đảo xâu ta vẫn nhận được chính xâu đó. Hãy liệt kê tất cả các xâu thuận nghịch độc có độ dài n và ghi lại những xâu đó vào tệp thuang.out theo từng dòng, dòng đầu tiên ghi lại giá trị của n, các dòng tiếp theo là những xâu thuận nghịch độc có độ dài n. Ví dụ: với n = 4, ta có được những xâu thuận nghịch độc có dạng sau:

```
4
0   0   0   0
0   1   1   0
1   0   0   1
1   1   1   1
```

4.2. Viết chương trình quản lý điểm thi của sinh viên bằng single (double) link list bao gồm những thao tác sau:

- Nhập dữ liệu;
- Hiển thị dữ liệu theo lớp, xếp loại ...;
- Sắp xếp dữ liệu;
- Tìm kiếm dữ liệu;
- In ấn kết quả.

Trong đó, thông tin về mỗi sinh viên được định nghĩa thông qua cấu trúc sau:

```
typedef struct {
    int masv;                    // mã sinh viên;
```

```
    char malop[12];          // mã lớp
    char hoten[30];          // họ tên sinh viên
    float diemki;            // điểm tổng kết kỳ 1
    float diemkii;           // điểm tổng kết kỳ 2
    float diemtk;            // điểm tổng kết cả năm
    char xeploai[12];        // xếp loại
} sinhvien;
```

4.3. **Biểu diễn biểu thức theo cú pháp Ba Lan.** Biểu thức nguyên là một dãy được thành lập từ các biến kiểu nguyên nối với nhau bằng các phép toán hai ngôi (cộng: + , trừ: - , nhân: *) và các dấu mở ngoặc đơn '(', đóng ngoặc đơn ')'. Nguyên tắc đặt tên biến và thứ tự thực hiện các phép toán được thực hiện như sau:

Qui tắc đặt tên biến: Là dãy các kí tự chữ in thường hoặc kí tự số độ dài không quá 8, kí tự bắt đầu phải là một chữ cái.

Qui tắc thực hiện phép toán: Biểu thức trong ngoặc đơn được tính trước, phép toán nhân '*' có độ ưu tiên cao hơn so với hai phép toán cộng và trừ. Hai phép toán cộng '+' và trừ có cùng độ ưu tiên. Ví dụ: a * b + c phải được hiểu là: (a * b) + c.

Dạng viết không ngoặc Ba Lan cho biểu thức nguyên được định nghĩa như sau:

- Nếu e là tên biến thì dạng viết Ba Lan của nó chính là e.
- Nếu e_1 và e_2 là hai biểu thức có dạng viết Ba Lan tương ứng là d_1 và d_2 thì dạng viết Ba Lan của $e_1 + e_2$ là d_1 d_2+, của $e_1 - e_2$ là d_1 d_2-, của e_1* e_2 là d_1 d_2* (Giữa d_1 và d_2 có đúng một dấu cách, trước dấu phép toán không có dấu cách).
- Nếu e là biểu thức có dạng viết Ba Lan là d thì dạng viết Ba Lan của biểu thức có ngoặc đơn (e) chính là d (không còn dấu ngoặc nữa). Ví dụ: Biểu thức (c + b*(f - d)) có dạng viết Ba Lan là: c b f d -*+.

Cho tệp dữ liệu balan.in được tổ chức thành từng dòng, mỗi dòng không dài quá 80 ký tự là biểu diễn của biểu thức nguyên A. Hãy dịch các biểu thức nguyên A thành dạng viết Ba Lan của A ghi vào tệp balan.out theo từng dòng. Ví dụ: với tệp balan.in dưới đây sẽ cho ta kết quả như sau:

balan.in	balan.out
a+b	a b+
a -b	a b -
a*b	a b*
(a - b) +c	a b - c+
(a + b) * c	a b+ c*
(a + (b -c))	a b c -+
(a + b*(c -d))	a b c d -*+
((a + b) *c - (d + e) * f)	a b + c* d e+f* -

4.4. **Tính toán giá trị biểu thức Ba Lan.** Cho tệp dữ liệu balan.in gồm 2*n dòng trong đó, dòng có số thứ tự lẻ (1, 3, 5, ...) ghi lại một xâu là biểu diễn Ba Lan của biểu thức nguyên A, dòng có số thứ tự chẵn (2, 4, 6, . .) ghi lại giá trị của các biến xuất hiện trong A. Hãy tính giá trị của biểu thức A, ghi lại giá trị của A vào tệp balan.out từng dòng theo thứ tự: Dòng có thứ tự lẻ ghi lại biểu thức Ba Lan của A sau khi đã thay thế các giá trị tương ứng của biến trong A, dòng có thứ tự chẵn ghi lại giá trị của biểu thức A.

Ví dụ với tệp balan.in dưới đây sẽ cho ta kết quả như sau:

balan.in	balan.out
a b+	3 5+
3 5	8
a b -	7 3 -

7 3	4
a b*	4 3 *
4 3	12
c a b -+	3 4 5 -+
3 4 5	2

4.5. **Lập lịch với mức độ ưu tiên.** Để lập lịch cho CPU đáp ứng cho các quá trình đang đợi của hệ thống, người ta biểu diễn mỗi quá trình bằng một bản ghi bao gồm những thông tin: số quá trình(Num) là một số tự nhiên nhỏ hơn 1024, tên quá trình (Proc) là một xâu ký tự độ dài không quá 32 không chứa dấu trống ở giữa, độ ưu tiên quá trình là một số nguyên dương (Pri) nhỏ hơn 10, thời gian thực hiện của quá trình (Time) là một số thực. Các quá trình đang đợi trong hệ được CPU đáp ứng thông qua một hàng đợi được gọi là hàng đợi các quá trình, hàng đợi các quá trình với độ ưu tiên được xây dựng sao cho những điều kiện sau được thoả mãn:

- Các quá trình được sắp theo thứ tự ưu tiên;
- Đối với những quá trình có cùng độ ưu tiên thì quá trình nào có thời gian thực hiện ít nhất được xếp lên trước nhất.

Cho tệp dữ liệu lich.in được tổ chức như sau:

- Dòng đầu tiên ghi lại một số tự nhiên n là số các quá trình;
- n dòng kế tiếp, mỗi dòng ghi lại thông tin về một quá trình đang đợi.

Hãy xây dựng hàng đợi các quá trình với độ ưu tiên. Ghi lại thứ tự các quá trình mà CPU đáp ứng trên một dòng của tệp lich.out, mỗi quá trình được phân biệt với nhau bởi một hoặc vài ký tự trống, dòng kế tiếp ghi lại số giờ cần thiết mà CPU cần đáp ứng cho các quá trình. Ví dụ với tệp lich.in dưới đây sẽ cho ta kết quả như sau:

lich.in

7

1	Data_Processing	1	10
2	Editor_Program	1	20
3	System_Call	3	0.5
4	System_Interative	3	1
5	System_Action	3	2
6	Writing_Data	2	20
7	Reading_Data	2	10

lich.out

3 4 5 7 6 1 2

63.5

4.6. **Thuật toán RR (Round Robin).** Thuật toán SJF đáp ứng được tối đa các quá trình hoạt động trong hệ, tuy nhiên sẽ có nhiều quá trình có chi phí thời gian lớn phải đợi nhiều quá trình có chi phí thời gian nhỏ thực hiện. Với thuật toán SJF, tính công bằng của hệ bị vi phạm. Để khắc phục điều trên, thuật toán Round Robin thực hiện chọn một lượng tử thời gian thích hợp, sau đó đáp ứng cho mỗi quá trình theo từng vòng với lượng tử thời gian đã chọn. Ưu điểm của RR là tính công bằng của hệ được đảm bảo, số các quá trình được CPU đáp ứng trên một đơn vị thời gian chấp nhận được. Nhược điểm lớn nhất của thuật toán là việc lựa chọn lượng tử thời gian đáp ứng cho mỗi quá trình sao cho tối ưu không phải là đơn giản. Hãy viết chương trình mô phỏng thuật toán lập lịch RR.

4.7. Quản lý bộ nhớ (Memory Management)

Quản lý bộ nhớ là quá trình điều khiển việc nạp các quá trình vào bộ nhớ. Bộ nhớ có thể được coi là một mảng một chiều hoặc

nhiều chiều, mỗi ô nhớ được xác định thông qua địa chỉ của nó. Địa chỉ của ô nhớ phụ thuộc vào phương pháp quản lý bộ nhớ, các phương pháp quản lý bộ nhớ thông dụng hiện nay là phân trang (Paging) hoặc phân đoạn (Segmentation). Ở mỗi thời điểm, có thể có nhiều quá trình được nạp vào những vùng bộ nhớ khác nhau làm cho bộ nhớ có những lỗ hổng dư thừa gọi là Hole. Bài toán đặt ra là làm thế nào để có thể nạp tiếp các quá trình vào các Hole bộ nhớ. Những thuật toán cơ bản sau sẽ giúp chúng ta minh họa điều đó:

Thuật toán First Fit: Duyệt theo danh sách các lỗ hổng bộ nhớ, chọn lỗ hổng bộ nhớ đầu tiên có thể để thực hiện việc nạp chương trình. Ưu điểm của phương pháp này là phương pháp cài đặt đơn giản, tốc độ nhanh. Nhược điểm lớn nhất của nó là quá trình nạp có thể tạo nên các lỗ hổng bộ nhớ mịn hơn.

Thuật toán Best Fit: Duyệt theo danh sách các lỗ hổng bộ nhớ, tìm lỗ hổng bộ nhớ có kích cỡ thích hợp nhất để thực hiện nạp chương trình. Ưu điểm của phương pháp này là các lỗ hổng bộ nhớ được sử dụng triệt để. Nhược điểm của nó là cài đặt phức tạp, quá trình duyệt có thể mất nhiều thời gian.

Thuật toán Worsd Fit: Luôn lấy lỗ hổng bộ nhớ lớn nhất để thực hiện việc nạp chương trình. Ưu điểm của nó là đơn giản, dễ cài đặt. Nhược điểm của nó là gây lãng phí bộ nhớ.

Comapaction (làm đặc bộ nhớ): Làm đặc bộ nhớ là phương pháp dịch chuyển các quá trình đã được nạp vào bộ nhớ tại các miền thích hợp, sao cho các lỗ hổng bộ nhớ được đặt cạnh nhau để tạo nên một lỗ hổng bộ nhớ lớn nhất có thể được. Hãy viết chương trình mô phỏng lại những thuật toán quản lý bộ nhớ trên.

CHƯƠNG 5

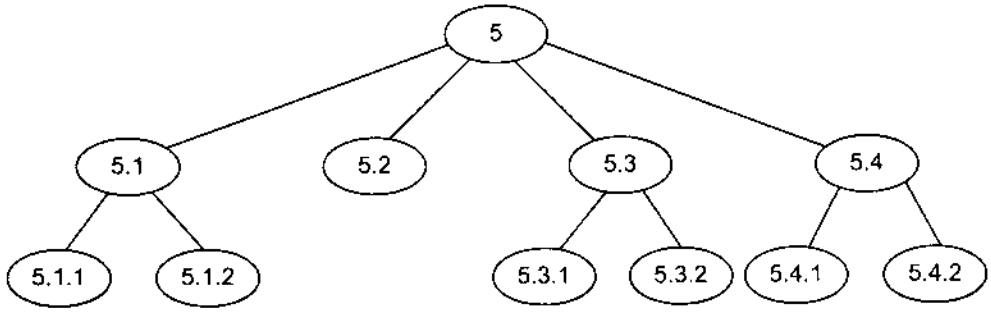
CẤU TRÚC DỮ LIỆU CÂY (TREE)

5.1. ĐỊNH NGHĨA VÀ KHÁI NIỆM

Cây là một tập hợp hữu hạn các node có cùng chung một kiểu dữ liệu, trong đó có một node đặc biệt gọi là node gốc (root). Giữa các node có một quan hệ phân cấp gọi là "quan hệ cha con". Có thể định nghĩa một cách đệ qui về cây như sau:

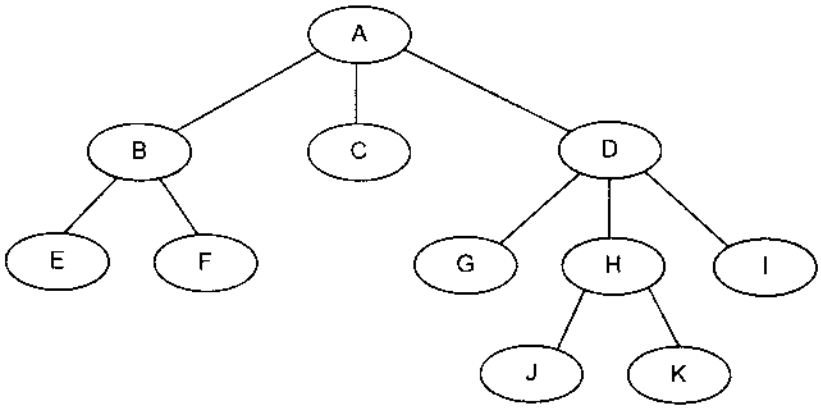
- Một node là một cây. Node đó cũng là gốc (root) của cây ấy.
- Nếu n là một node và T_1, T_2, ..., T_k là các cây với n_1, n_2, ..., n_k lần lượt là gốc thì một cây mới T sẽ được tạo lập bằng cách cho node n trở thành cha của các node n_1, n_2, ..., n_k hay node n trở thành gốc và T_1, T_2, ..., T_k là các cây con (subtree) của gốc.

Ví dụ: Cấu trúc tổ chức thư mục (directory) của DOS là một cấu trúc cây.



Hình 5.1. Ví dụ về một cây thư mục

Một cây được gọi là rỗng nếu nó không có bất kỳ một node nào. Số các node con của một node được gọi là cấp (degree) của node đó. Ví dụ: trong cây ở hình 5.2, cấp của node A là 3, cấp của node B là 2, cấp của node D là 3, cấp của node H là 2.



Hình 5.2 Phân cấp cây thư mục

Node có cấp bằng 0 được gọi là lá (leaf) hay node tận cùng (terminal node). Ví dụ: các node E, F, C, G, I, J, K được gọi là lá. Node không là lá được gọi là node trung gian hay node nhánh (branch node). Ví dụ node B, D, H là các node nhánh.

Cấp cao nhất của node trên cây gọi là cấp của cây, trong trường hợp cây ở hình 5.2 cấp của cây là 3.

Gốc của cây có số mức là 1. Nếu node cha có số mức là i thì node con có số mức là i + 1. Ví dụ gốc A có số mức là 1, D có số mức là 2, G có số mức là 3, j có số mức là 4.

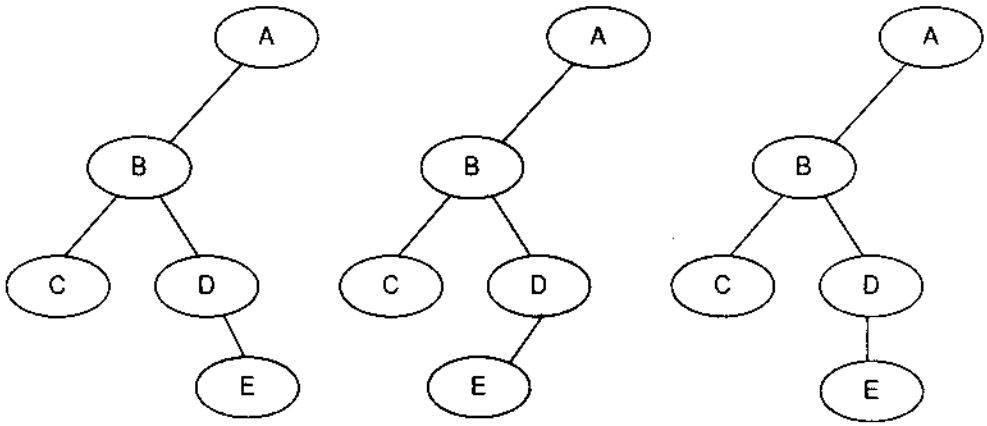
Chiều cao (height) hay chiều sâu (depth) của một cây là số mức lớn nhất của node trên cây đó. Cây hình 5.2 có chiều cao là 4.

Đường đi từ node n_1 đến n_k là dãy các node $n_1, n_2, ..., n_k$ sao cho n_i là node cha của node n_{i+1} ($1 \le i < k$), độ dài của đường đi (path length) được tính bằng số các node trên đường đi trừ đi 1 vì nó phải tính từ node bắt đầu và node kết thúc. Ví dụ: trong cây hình 5.2 đường đi từ node A tới node G là 2, đường đi từ node A đến node K là 3.

Một cây được gọi là có thứ tự nếu chúng ta xét đến thứ tự các cây con trong cây (ordered tree), ngược lại là cây không có thứ tự (unordered tree). Thông thường các cây con được tính theo thứ tự từ trái sang phải.

5.2. CÂY NHỊ PHÂN

Cây nhị phân là một dạng quan trọng của cấu trúc cây có đặc điểm là mọi node trên cây chỉ có tối đa là hai node con. Cây con bên trái của cây nhị phân được gọi là left subtree, cây con bên phải của cây được gọi là right subtree. Đối với cây nhị phân, bao giờ cũng được phân biệt cây con bên trái và cây con bên phải. Như vậy, cây nhị phân là một cây có thứ tự. Ví dụ trong hình 5.3, các cây đều là cây nhị phân.

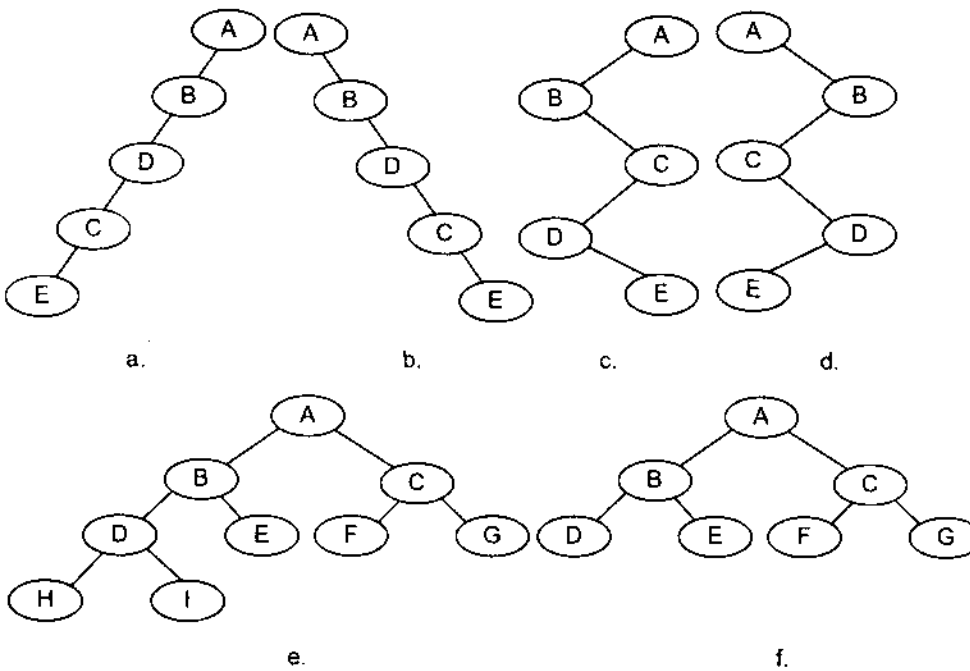


Hình 5.3 Các cây nhị phân

Các cây nhị phân có dạng đặc biệt bao gồm:

- **Cây nhị phân lệch trái** (hình 5.4a): là cây nhị phân chỉ có các node bên trái.
- **Cây nhị phân lệnh phải** (hình 5.4b): là cây chỉ bao gồm các node phải.
- **Cây nhị phân zic zắc** (hình 5.4c, 5.4d): node trái và node phải của cây đan xen nhau thành một hình zic zắc.
- **Cây nhị phân hoàn chỉnh** (strictly binary tree: hình 5.4e): một cây nhị phân được gọi là hoàn chỉnh nếu như node gốc và tất cả các node trung gian đều có hai con.

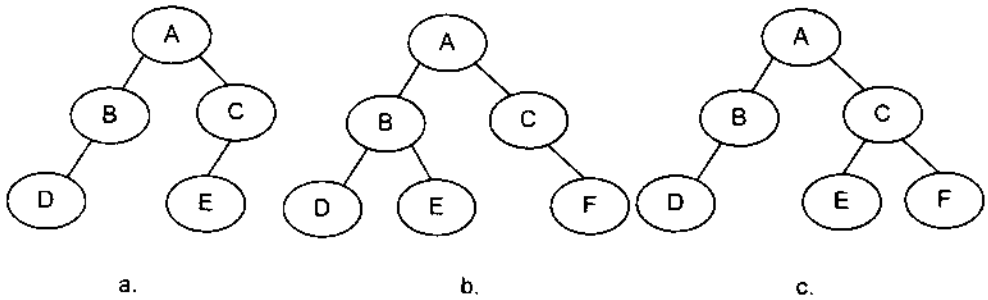
- **Cây nhị phân đầy đủ** (complete binary tree: hình 5.4f): Một cây nhị phân được gọi là đầy đủ với chiều sâu d thì nó phải là cây nhị phân hoàn chỉnh và tất cả các node lá đều có chiều sâu là d.



Hình 5.4 Các cây nhị phân có dạng đặc biệt

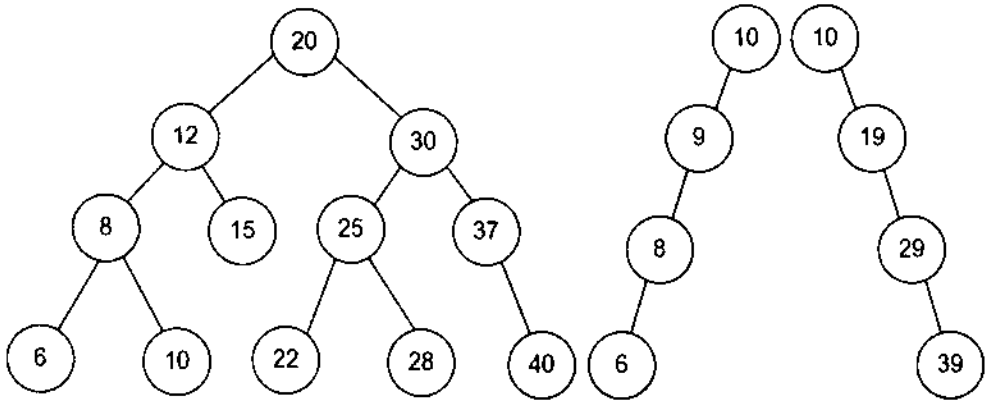
- **Cây nhị phân hoàn toàn cân bằng** (hình 5.5): là cây nhị phân mà ở tất cả các node của nó số node trên nhánh cây con bên trái và số node trên nhánh cây con bên phải chênh lệnh nhau không quá 1. Nếu ta gọi N_l là số node của nhánh cây con bên trái và N_r là số node của nhánh cây con bên phải, khi đó cây nhị phân hoàn toàn cân bằng chỉ có thể ở một trong 3 trường hợp:
  - Số node nhánh cây con bên trái bằng số node nhánh cây con bên phải bằng $N_l = N_r$ (hình 5.5a).
  - Số node nhánh cây con bên trái bằng số node nhánh cây con bên phải cộng 1: $N_l = N_r + 1$(hình 5.5b)

- Số node nhánh cây con bên trái bằng số node nhánh cây con bên phải trừ 1: $N_l = N_r - 1$ (hình 5.5c).



Hình 5.5 Cây nhị phân hoàn toàn cân bằng

- **Cây nhị phân tìm kiếm**: là một cây nhị phân hoặc bị rỗng hoặc tất cả các node trên cây thỏa mãn điều kiện sau:
  - Nội dung của tất cả các node thuộc nhánh cây con bên trái đều nhỏ hơn nội dung của node gốc.
  - Nội dung của tất cả các node thuộc nhánh cây con bên phải đều lớn hơn nội dung của node gốc.
  - Cây con bên trái và cây con bên phải cũng tự nhiên hình thành hai cây nhị phân tìm kiếm.



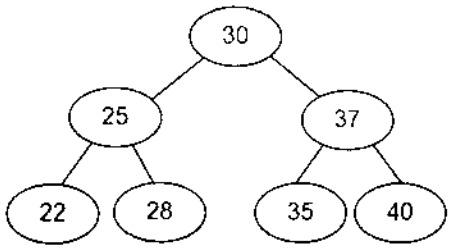
Hình 5.6. Ví dụ về cây nhị phân tìm kiếm

5.3. BIỂU DIỄN CÂY NHỊ PHÂN

5.3.1. Biểu diễn cây nhị phân bằng danh sách tuyến tính

Trong trường hợp cây nhị phân đầy đủ, ta có thể dễ dàng biểu diễn cây nhị phân bằng một mảng lưu trữ kế tiếp. Trong đó node gốc là phần tử đầu tiên của mảng (phần tử thứ 1), node con thứ i ≥ 1 của cây nhị phân là phần tử thứ 2i, 2i + 1 hay cha của node thứ j là [j/2]. Với qui tắc đó, cây nhị phân có thể biểu diễn bằng một vector V sao cho nội dung của node thứ i được lưu trữ trong thành phần V[i] của vector V. Ngược lại, nếu biết địa chỉ của phần tử thứ i trong vector V chúng ta cũng hoàn toàn xác định được ngược lại địa chỉ của node cha, địa chỉ node gốc trong cây nhị phân.

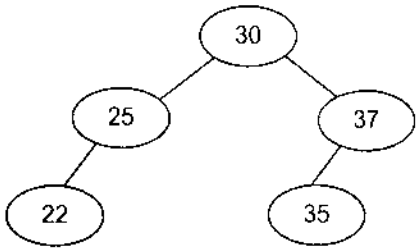
Ví dụ: Cây nhị phân được lưu trữ kế tiếp như trong hình 5.7.



V[0]	V[1]	V[2]	V[3]	V[4]	V[5]	V[6]
30	25	37	22	28	35	40

Hình 5.7. Lưu trữ kế tiếp của cây nhị phân

Đối với cây nhị phân không đầy đủ, việc lưu trữ bằng mảng tỏ ra không hiệu quả vì chúng ta phải bỏ trống quá nhiều phần tử gây lãng phí bộ nhớ (xem ví dụ trong hình 5.8).



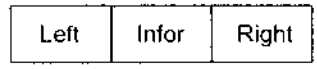
V[0]	V[1]	V[2]	V[3]	V[4]	V[5]	V[6]
30	25	37	22	ϕ	35	ϕ

Hình 5.8. Lưu trữ kế tiếp của cây nhị phân không đầy đủ

5.3.2. Biểu diễn cây nhị phân bằng danh sách móc nối

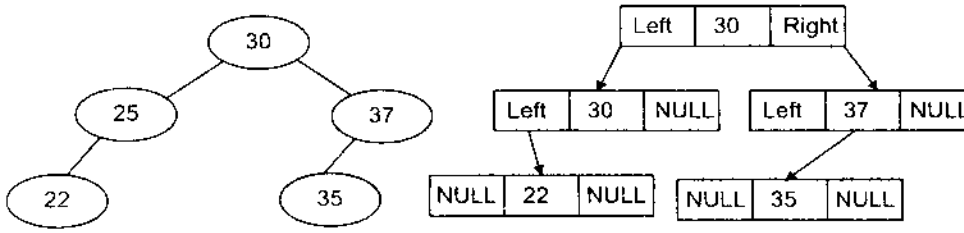
Trong cách lưu trữ cây nhị phân bằng danh sách móc nối, mỗi node được mô tả bằng ba loại thông tin chính: Left, infor, right.

Left là một con trỏ trỏ tới node bên trái của cây nhị phân; infor: là thông tin về node, infor có thể là một biến đơn hoặc một cấu trúc; right là một con trỏ trỏ tới node bên phải của cây nhị phân. Trong trường hợp node là node lá thì con trỏ left và con trỏ right được trỏ tới con trỏ NULL. Đối với node lệch trái, con trỏ right sẽ trỏ tới con trỏ NULL, ngược lại đối với node lệch phải, con trỏ left cũng sẽ trỏ tới con trỏ NULL. Cấu trúc của một node được mô tả trong hình 5.9.



Hình 5.9 Mô tả một node của cây nhị phân.

Ví dụ: Cây nhị phân trong hình 5.10 sẽ được biểu diễn bằng danh sách liên kết như sau:



Hình 5.10 Biểu diễn cây nhị phân bằng danh sách móc nối

5.4. CÁC THAO TÁC TRÊN CÂY NHỊ PHÂN

5.4.1. Định nghĩa cây nhị phân bằng danh sách tuyến tính

Mỗi node trong cây được khai báo như một cấu trúc gồm 3 trường: infor, left, right. Toàn bộ cây có thể coi như một mảng mà mỗi phần tử của nó là một node. Trường infor tổng quát có thể là một đối tượng dữ liệu kiểu cơ bản hoặc một cấu trúc. Ví dụ: định nghĩa một cây nhị phân lưu trữ danh sách các số nguyên:

```
#define MAX 100
#define TRUE  1
#define FALSE 0
struct node {
    int infor;
    int left;
    int right;
};
typedef struct   node node[MAX];
```

5.4.2. Định nghĩa cây nhị phân theo danh sách liên kết:

```
struct node {
int infor;
struct node *left;
struct node *right;
}
typedef struct   node *NODEPTR
```

5.4.3. Các thao tác trên cây nhị phân

Cấp phát bộ nhớ cho một node mới của cây nhị phân:

```
NODEPTR Getnode(void) {
    NODEPTR p;
    p = (NODEPTR) malloc(sizeof(struct node));
    return(p);
}
```

Giải phóng node đã được cấp phát:

```
void Freenode(NODEPTR p) {
    free(p);
}
```

Khởi động cây nhị phân:

```
void Initialize(NODEPTR *ptree) {
    *ptree = NULL;
}
```

Kiểm tra tính rỗng của cây nhị phân:

```
int Empty(NODEPTR *ptree) {
if(*ptree = NULL)
    return(TRUE);
return(FALSE);
}
```

Tạo một node lá cho cây nhị phân:

- Cấp phát bộ nhớ cho node;
- Gán giá trị thông tin thích hợp cho node;
- Tạo liên kết cho node lá;

```
NODEPTR  Makenode(int x) {
    NODEPTR p;
    p = Getnode();          // cấp phát bộ nhớ cho node
    p -> infor = x;         // gán giá trị thông tin thích hợp
    p -> left = NULL;       // tạo liên kết trái của node lá
    p -> right = NULL;      // tạo liên kết phải của node lá
    return(p);
}
```

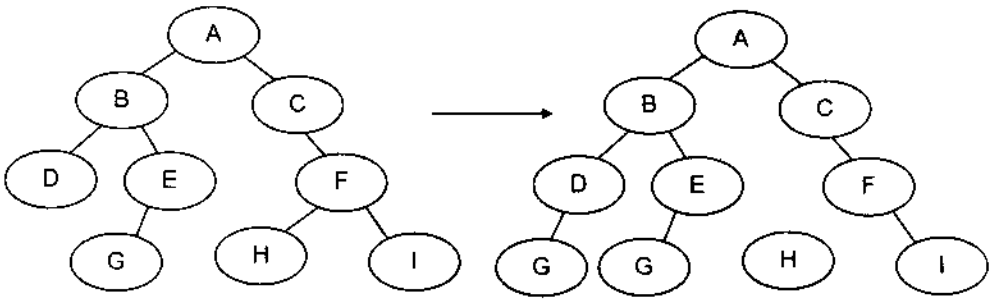
Tạo node con bên trái của cây nhị phân:

Để tạo được node con bên trái là node lá của node p, chúng ta thực hiện như sau:

- Nếu node p không có thực (p == NULL), ta không thể tạo được node con bên trái của node p;

- Nếu node p đã có node con bên trái (p -> left != NULL), thì chúng ta cũng không thể tạo được node con bên trái node p;
- Nếu node p chưa có node con bên trái, thì việc tạo node con bên trái chính là thao tác tạo node đã được xây dựng như trên;

Hình 5.11 sẽ minh họa cho thao tác tạo node con X phía bên trái của node D.



Hình 5.11 Mô tả thao tác thêm node con bên trái cây nhị phân

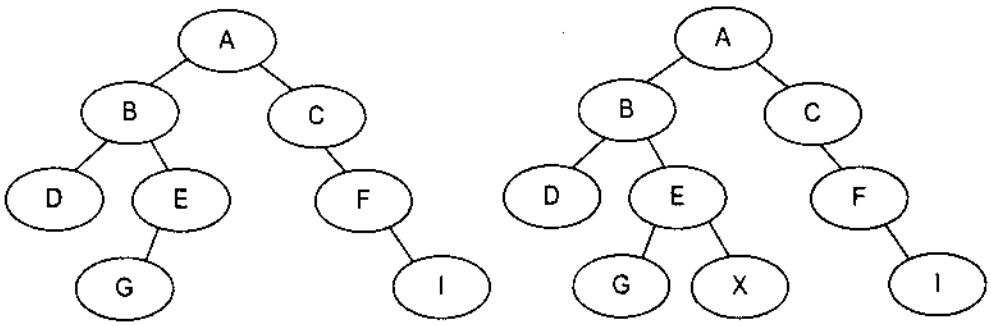
```
void Setleft(NODEPTR p, int x ) {
    if(p = NULL) {
            // Nếu node p không có thực thì không thể thực hiện được
        printf("\n Node p không có thực");
        delay(2000); return;
    }
/* Nếu node p có thực và tồn tại lá con bên trái thì cũng không thực hiện được */
    else if(p -> left != NULL) {
        printf("\n Node p đã có node con bên trái");
        delay(2000); return;
    }
    // Nếu node có thực và chưa có node trái
    else
        p -> left = Makenode(x);
}
```

Tạo node con bên phải của cây nhị phân:

Để tạo được node con bên phải là node lá của node p, chúng ta làm như sau:

- Nếu node p không có thực (p = NULL), thì ta không thể thực hiện được thao tác thêm node lá vào node phải node p;
- Nếu node p có thực (p != NULL) và đã có node con bên phải thì thao tác cũng không thể thực hiện được;
- Nếu node p có thực và chưa có node con bên phải thì việc tạo node con bên phải node p được thực hiện thông qua thao tác Makenode();

Hình 5.12 sẽ minh họa cho thao tác tạo node con X phía bên phải của node E.



Hình 5.12 Mô tả thao tác thêm node con bên phải cây nhị phân

```
void Setright(NODEPTR  p, int x ) {
    if(p = NULL) {                    // Nếu node p không có thực
        printf("\n Node p không có thực");
        delay(2000); return;
    }
                  // Nếu node p có thực & đã có node con bên phải
    else if(p -> right != NULL) {
```

```
            printf("\n Node p đã có node con bên phải");
            delay(2000); return;
        }
                // Nếu node p có thực & chưa có node con bên phải
        else
            p -> right = Makenode(x);
    }
```

Thao tác xoá node con bên trái cây nhị phân:

Thao tác loại bỏ node con bên trái node p được thực hiện như sau:

- Nếu node p không có thực thì thao tác không thể thực hiện;
- Nếu node p có thực (p == NULL) thì kiểm tra xem p có node lá bên trái hay không;
  - Nếu node p có thực và p không có node lá bên trái thì thao tác cũng không thể thực hiện được;
  - Nếu node p có thực (p != NULL) và có node con bên trái là q thì:
    - Nếu node q không phải là node lá thì thao tác cũng không thể thực hiện được (q -> left != NULL || q -> right != NULL);
    - Nếu node q là node lá (q -> left == NULL && q -> right == NULL) thì:
      - Giải phóng node q;
      - Thiết lập liên kết mới cho node p;

Thuật toán được thể hiện bằng thao tác Delleft() như dưới đây:

```
int Delleft(NODEPTR p) {
    NODEPTR q; int x;
```

```
if(p = NULL)
    printf("\n Node p không có thực"); delay(2000);
    exit(0);
}
q = p -> left;              // q là node cần xoá;
x = q -> infor;             // x là nội dung cần xoá
if(q = NULL) {              // kiểm tra p có lá bên trái hay không
    printf("\n Node p không có lá bên trái");
    delay(2000); exit(0);
}
if(q -> left != NULL || q -> right != NULL) {
    // kiểm tra q có phải là node lá hay không
    printf("\n q không là node lá");
    delay(2000); exit(0);
}
p -> left = NULL;           // tạo liên kết mới cho p
Freenode(q);                // giải phóng q
return(x);
}
```

Thao tác xoá node con bên phải cây nhị phân:

Thao tác loại bỏ node con bên phải node p được thực hiện như sau:

- Nếu node p không có thực thì thao tác không thể thực hiện;
- Nếu node p có thực (p = NULL) thì kiểm tra xem p có node lá bên phải hay không;
  - Nếu node p có thực và p không có node lá bên phải thì thao tác cũng không thể thực hiện được;
  - Nếu node p có thực (p != NULL) và có node con bên phải là q thì:

- Nếu node q không phải là node lá thì thao tác cũng không thể thực hiện được (q -> left != NULL || q -> right != NULL);
- Nếu node q là node lá (q -> left == NULL && q -> right == NULL) thì:
  - Giải phóng node q;
  - Thiết lập liên kết mới cho node p;

Thuật toán được thể hiện bằng thao tác Delright() như dưới đây:

```
int Delright(NODEPTR p) {
NODEPTR q; int x;
if(p = NULL)
    printf("\n Node p không có thực"); delay(2000);
    exit(0);
}
q = p -> right;             // q là node cần xoá;
    x = q -> infor;             // x là nội dung cần xoá
if(q = NULL) {              // kiểm tra p có lá bên phải hay không
    printf("\n Node p không có lá bên phải");
    delay(2000); exit(0);
}
if(q -> left != NULL || q -> right != NULL) {
// kiểm tra q có phải là node lá hay không
    printf("\n q không là node lá");
    delay(2000); exit(0);
}
p -> right = NULL;          // tạo liên kết cho p
Freenode(q);                // giải phóng q
return(x);
}
```

Thao tác tìm node có nội dung là x trên cây nhị phân:

Để tìm node có nội dung là x trên cây nhị phân, chúng ta có thể xây dựng bằng thủ tục đệ qui như sau:

- Nếu node gốc (proot) có nội dung là x thì proot chính là node cần tìm;
- Nếu proot = NULL thì không có node nào trong cây có nội dung là x;
- Nếu nội dung node gốc khác x (proot -> infor != x) và proot != NULL thì:
  - Tìm node theo nhánh cây con bên trái (proot = proot -> left);
  - Tìm theo nhánh cây con bên phải;

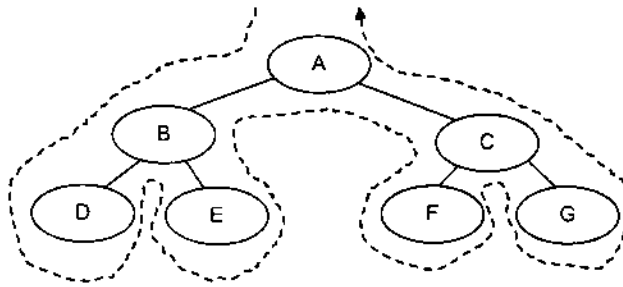
Thuật toán tìm một node có nội dung là x trong cây nhị phân được thể hiện như sau:

```
NODEPTR Search(NODEPTR proot, int x) {
    NODEPTR  p;
    if(proot -> infor = x)              // điều kiện dừng
        return(proot);
    if(proot = NULL)
        return(NULL);
    p = Search(proot -> left, x);       // tìm trong nhánh con bên trái
    if(p = NULL)                        // tìm trong nhánh con bên phải
        Search(proot -> right, x);
    return(p);
}
```

5.5. BA PHÉP DUYỆT CÂY NHỊ PHÂN (TRAVERSING BINARY TREE)

Phép duyệt cây là phương pháp thăm (visit) các node một cách có hệ thống sao cho mỗi node chỉ được thăm một lần. Có ba phương pháp để duyệt cây nhị phân đó là:

- Duyệt theo thứ tự trước (Preorder Traversal);
- Duyệt theo thứ tự giữa (Inorder Traversal);
- Duyệt theo thứ tự sau (Postorder Traversal).



Hình 5.13 Mô tả phương pháp duyệt cây nhị phân

5.5.1. Duyệt theo thứ tự trước (Preorder Traversal)

- Nếu cây rỗng thì không làm gì;
- Nếu cây không rỗng thì:
  - Thăm node gốc của cây;
  - Duyệt cây con bên trái theo thứ tự trước;
  - Duyệt cây con bên phải theo thứ tự trước;

Ví dụ: Với cây trong hình 5.13 thì phép duyệt Preorder cho ta kết quả duyệt theo thứ tự các node là: A -> B -> D -> E -> C -> F -> G.

Với cách duyệt theo thứ tự trước, chúng ta có thể cài đặt cho cây được định nghĩa trong mục 5.4 bằng một thủ tục đệ qui như sau:

```
void Pretraverse (NODEPTR proot ) {
    if(proot != NULL) {                 // nếu cây không rỗng
        printf("%d", proot -> infor);   // duyệt node gốc
        Pretravese(proot -> left);      // duyệt nhánh cây con bên trái
        Pretravese(proot -> right);     // duyệt nhánh con bên phải
    }
}
```

5.5.2. Duyệt theo thứ tự giữa (Inorder Traversal)

- Nếu cây rỗng thì không làm gì;
- Nếu cây không rỗng thì:
  - Duyệt cây con bên trái theo thứ tự giữa;
  - Thăm node gốc của cây;
  - Duyệt cây con bên phải theo thứ tự giữa;

Ví dụ: Cây trong hình 5.13 thì phép duyệt Inorder cho ta kết quả duyệt theo thứ tự các node là: D -> B -> E -> A -> F -> C -> G.

Với cách duyệt theo thứ tự giữa, chúng ta có thể cài đặt cho cây được định nghĩa trong mục 5.4 bằng một thủ tục đệ qui như sau:

```
void  Intravese (NODEPTR proot ) {
    if(proot != NULL) {                 // nếu cây không rỗng
        Intravese(proot -> left);       // duyệt nhánh cây con bên trái
        printf("%d", proot -> infor);   // duyệt node gốc
        Intravese(proot -> right);      // Duyệt nhánh con bên phải
    }
}
```

5.5.3. Duyệt theo thứ tự sau (Postorder Traversal)

- Nếu cây rỗng thì không làm gì;

- Nếu cây không rỗng thì:
  - Duyệt cây con bên trái theo thứ tự sau;
  - Duyệt cây con bên phải theo thứ tự sau;
  - Thăm node gốc của cây;

Ví dụ: Cây trong hình 5.13 thì phép duyệt Postorder cho ta kết quả duyệt theo thứ tự các node là:D -> E -> B -> F -> G -> C -> A .

Với cách duyệt theo thứ tự giữa, chúng ta có thể cài đặt cho cây được định nghĩa trong mục 5.4 bằng một thủ tục đệ qui như sau:

```
void Posttraverse (NODEPTR proot ) {
    if(proot != NULL) {                 // nếu cây không rỗng
        Posttravese(proot -> left);     // duyệt nhánh cây con bên trái
        Posttravese(proot -> right);    // duyệt nhánh con bên phải
        printf("%d", proot -> infor);   // duyệt node gốc
    }
}
```

5.6. CÀI ĐẶT CÂY NHỊ PHÂN BẰNG DANH SÁCH TUYẾN TÍNH

```
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#include <alloc.h>
#include <string.h>
#include <dos.h>
#define TRUE 1
#define FALSE 0
#define MAX 100
typedef struct    {
   int infor;
```

```
    int left;
    int right;
} nodetype;
nodetype node[MAX]; int avail;
int Getnode(void) {
    int p;
    if(avail == -1) {
        printf("\n Hết node danh sách");
        return(avail);
    }
    p = avail;
    avail = node[avail].left;
    return(p);
}
void Freenode (int p) {
    node[p].left = avail;
    avail = p;
}
void Initialize(int *ptree) {
    *ptree = -1;
}
int Empty(int *ptree) {
    if(*ptree == -1)
        return(TRUE);
    return(FALSE);
}
int Makenode(int x) {
    int p;
```

```
    p = Getnode();
    node[p].infor = x;
    node[p].left = -1;
    node[p].right = -1;
    return(p);
}
void Setleft(int p, int x) {
    if(p == -1)
        printf("\n Node p không có thực");
    else {
        if(node[p].left != -1)
            printf("\n Node p đã có bên trái");
        else
            node[p].left = Makenode(x);
    }
    delay(2000);
}
void Setright(int p, int x) {
    if(p == -1)
        printf("\n Node p không có thực");
    else {
        if(node[p].right != -1)
            printf("\n Node p đã có bên phải");
        else
            node[p].right = Makenode(x);
    }
    delay(2000);
}
```

```
int Delleft(int p) {
    int q, x;
    if(p == -1) {
        printf("\n Node p không có thực");
        delay(2000); return(-1);
    }
    else {
        q = node[p].left;
        x = node[q].infor;
        if(q = -1) {
            printf("\n Node p không có nút trái");
            delay(2000); return(q);
        }
        else if(node[q].left != -1 || node[q].right != -1) {
                printf("\n Q không là node lá");
                delay(2000); return(-1);
        }
    }
    node[p].left = -1;
    Freenode(q); return(x);
}
int Delright (int p) {
    int q, x;
    if(p == -1) {
        printf("\n Node p không có thực");
        delay(2000); return(-1);
    }
    else {
```

```
        q = node[p].right;
        x = node[q].infor;
        if(q == -1) {
            printf("\n Node p không có node trái");
            delay(2000); return(q);
        }
        else if(node[q].left != -1 || node[q].right != -1) {
                printf("\n Q không là node lá");
                delay(2000); return(-1);
        }
    }
    node[p].right = -1;
    Freenode(q); return(x);
}
void Pretrav(int proot) {
    if(proot != -1) {
        printf("%5d", node[proot].infor);
        Pretrav(node[proot].left);
        Pretrav(node[proot].right);
    }
}
void Intrav(int proot) {
    if(proot != -1) {
        Intrav(node[proot].left);
        printf("\n %d", node[proot].left);
        Intrav(node[proot].right);
    }
}
```

```
void Postrav(int proot) {
    if(proot != -1) {
        Postrav(node[proot].left);
        Postrav(node[proot].right);
        printf("\n %d", node[proot].left);
    }
}
int Search(int proot, int x) {
    int p;
    if(node[proot].infor == x)
        return(proot);
    if(proot == -1)
        return(-1);
    p = Search(node[proot].left, x);
    if(p == -1)
        p = Search(node[proot].right, x);
    return(p);
}
void Cleartree(int proot) {
    if(proot != -1) {
        Cleartree(node[proot].left);
        Cleartree(node[proot].right);
        Freenode(proot);
    }
}
void main(void ) {
    int i, noidung, noidung1, chucnang, p, ptree;
    char c; clrscr(); avail = 0;
```

```
for (i = 0; i<MAX; i ++ )
    node[i].left = i + 1;
node[MAX-1].left = -1;
Initialize(&ptree);
do {
    clrscr();
    printf("\n CÂY NHỊ PHÂN");
    printf("\n 1 - Tạo node gốc của cây");
    printf("\n 2 - Thêm một node lá bên trái");
    printf("\n 3 - Thêm một node lá bên phải");
    printf("\n 4 - Xóa một node lá bên trái");
    printf("\n 5 - Xóa một node lá bên phải");
    printf("\n 6 - Duyệt cây theo thứ tự trước");
    printf("\n 7 - Duyệt cây theo thứ tự giữa");
    printf("\n 8 - Duyệt cây theo thứ tự sau");
    printf("\n 9 - Tìm kiếm trên cây");
    printf("\n 10 - Xóa toàn bộ cây");
    printf("\n 0 - Kết thúc chương trình");
    printf("\n Chức năng lựa chọn:"); scanf("%d", &chucnang);
    switch(chucnang) {
        case 1:
            if(!Empty(&ptree))
                printf("\n Cây đã có node gốc");
            else {
                printf("\n Nội dung node gốc:");
                scanf("%d", &noidung);
                ptree = Makenode(noidung);
            }
```

```
        delay(1000); break;
    case 2:
        if(Empty(&ptree))
            printf("\n Cây chưa có gốc");
        else {
            printf("\n Node lá cần thêm:");
            scanf("%d", &noidung);
            p = Search(ptree, noidung);
            if(p != -1)
                printf("\n Nội dung bị trùng");
            else {
                printf("\n Nội dung node cha:");
                scanf("%d", &noidung1);
                p = Search(ptree, noidung1);
                if(p == -1)
                 printf("\n Không thấy node cha");
                else
                    Setleft(p, noidung);
            }
        }
        delay(2000); break;
    case 3:
        if(Empty(&ptree))
            printf("\n Cây chưa có gốc");
        else {
            printf("\n Node lá cần thêm:");
            scanf("%d", &noidung);
            p = Search(ptree, noidung);
```

```
        if(p != -1)
            printf("\n Nội dung bị trùng");
        else {
            printf("\n Nội dung node cha:");
            scanf("%d", &noidung1);
            p = Search(ptree, noidung1);
            if(p == -1)
             printf("\n Không thay node cha");
            else
                Setright(p, noidung);
        }
    }
    delay(2000); break;
case 4:
    printf("\n Nội dung node cha:");
    scanf("%d", &noidung);
    p = Search(ptree, noidung);
    if(p == -1)
        printf("\n Không thay node cha");
    else
        Delleft(p);
    delay(2000); break;
case 5:
    printf("\n Nội dung node cha:");
    scanf("%d", &noidung);
    p = Search(ptree, noidung);
    if(p == -1)
        printf("\n Không thấy node cha");
```

```
        else
            Delright(p);
        delay(2000); break;
    case 6:
        printf("\n Duyệt cây theo NLR:\n");
        if(Empty(&ptree))
            printf("Cây bị rỗng");
        else
            Pretrav(ptree);
        delay(2000); break;
    case 7:
        printf("\n Duyệt cây theo LNR:\n");
        if(Empty(&ptree))
            printf("Cây bị rỗng");
        else
            Intrav(ptree);
        delay(2000); break;
    case 8:
        printf("\n Duyệt cây theo LRN:\n");
        if(Empty(&ptree))
            printf("Cây bị rỗng");
        else
            Postrav(ptree);
        delay(2000); break;
    case 9:
        printf("\n Nội dung cần tìm:");
        scanf("%d", &noidung);
        if(Search(ptree, noidung) != -1)
```

```
            printf("\n Tìm thấy");
        else
            printf("\n Không tìm thấy");
        delay(2000); break;
    case 10:
        Cleartree(ptree);
        printf("OK!!:");
        delay(2000); break;
    }
  } while (chucnang != 0);
  Cleartree(ptree); ptree == -1;
}
```

5.7. CÀI ĐẶT CÂY NHỊ PHÂN HOÀN TOÀN CÂN BẰNG BẰNG LINK LIST

Vì cây nhị phân hoàn toàn cân bằng có số node nhánh cây con bên trái và số node nhánh cây con bên phải chênh lệnh nhau không quá 1, nên chúng ta định nghĩa thêm một trường là sonut để chỉ số node trên các nhánh cây:

Định nghĩa cây nhị phân hoàn toàn cân bằng:

```
struct node {
int infor;
int sonut;
struct node *left;
struct node *right;
};
typedef struct node *NODEPTR;
```

Thao tác chèn một node (Insrertion Node):

Các thao tác khác với cây nhị phân hoàn toàn cân bằng cũng giống như cây nhị phân. Riêng thao tác thêm node vào cây nhị phân hoàn toàn cân bằng sao cho cây vẫn đảm bảo tính cân cân bằng, chúng ta coi node mới thêm vào sẽ là node lá, node cha của node mới có sonut là 1 hoặc 2 và được điều khiển như sau:

- Xét con trỏ p tại gốc của cây nếu: p -> sonode > 2 thì cho p đi xuống: nếu p là lẻ cho p qua nhánh trái p = p -> left. Nếu p -> sonut là chẵn cho p qua nhánh phải: p = p -> right
- Nếu p -> sonut = 1: thêm node mới là nút con bên trái của p
- Nếu p -> sonut = 2: thêm node mới là nút con bên phải của p

Giải thuật thêm node vào cây nhị phân hoàn toàn đầy đủ được thể hiện như sau:

```
/* Thêm node x vào cây nhị phân hoàn toàn cân bằng */
void Insert(NODEPTR proot, int x) {
    NODEPTR p;
    p = proot;
    while(p -> sonut != 1 && p -> sonut != 2) {
        if(p -> sonut%2 = 1) {
            p -> sonut ++ ;
            p = p -> left;
        }
        else {
            p -> sonut ++ ;
            p = p -> right;
        }
    }
```

```
        if(p -> sonut == 1) {
            p -> sonut ++ ;
            Setleft(p, x);
        }
        else {
            p -> sonut ++ ;
            Setright(p, x);
        }
    }
```

Chương trình cài đặt cây nhị phân hoàn toàn cân bằng được thể hiện như sau:

```
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#include <alloc.h>
#include <string.h>
#include <dos.h>
#define TRUE 1
#define FALSE 0
#define MAX 100
struct node {
    int infor;
    int sonut;
    struct node *left;
    struct node *right;
};
typedef struct node *NODEPTR;
NODEPTR Getnode(void) {
```

```
    NODEPTR p;
    p = (NODEPTR)malloc(sizeof(struct node));
    return(p);
}
void Freenode(NODEPTR p) {
    free(p);
}
void Initialize(NODEPTR *ptree) {
    *ptree = NULL;
}
NODEPTR Makenode(int x) {
    NODEPTR p;
    p = Getnode();
    p -> infor = x;
    p -> sonut = 1;
    p -> left = NULL;
    p -> right = NULL;
    return(p);
}
void Setleft(NODEPTR p, int x) {
    if(p == NULL)
        printf("\n Node p không có thực");
    else {
        if(p -> left != NULL)
            printf("\n Node con bên trái đã tồn tại");
        else
            p -> left = Makenode(x);
    }
```

```
    delay(2000);
}
void Setright(NODEPTR p, int x) {
    if(p == NULL)
        printf("\n Node p không có thực");
    else {
        if(p -> right != NULL)
            printf("\n Node con bên phải đã tồn tại");
        else
            p -> right = Makenode(x);
    }
    delay(2000);
}
void Pretrav(NODEPTR proot) {
    if(proot != NULL) {
        printf("%5d", proot -> infor);
        Pretrav(proot -> left);
        Pretrav(proot -> right);
    }
}
void Intrav(NODEPTR proot) {
    if(proot != NULL) {
        Intrav(proot -> left);
        printf("%5d", proot -> infor);
        Intrav(proot -> right);
    }
}
void Postrav(NODEPTR proot) {
```

```
    if(proot != NULL) {
        Postrav(proot -> left);
        Postrav(proot -> right);
        printf("%5d", proot -> infor);
    }
}
void Insert(NODEPTR proot, int x) {
    NODEPTR p;
    p = proot;
    while(p -> sonut != 1 && p -> sonut != 2) {
        if(p -> sonut%2 == 1) {
            p -> sonut ++ ;
            p = p -> left;
        }
        else {
            p -> sonut ++ ;
            p = p -> right;
        }
    }
    if(p -> sonut == 1) {
        p -> sonut ++ ;
        Setleft(p, x);
    }
    else {
        p -> sonut ++ ;
        Setright(p, x);
    }
}
```

```
NODEPTR Search(NODEPTR proot, int x) {
    NODEPTR p;
    if(proot -> infor == x)
        return(proot);
    if(proot == NULL)
        return(NULL);
    p = Search(proot -> left, x);
    if(p == FALSE)
        p = Search(proot, x);
    return(p);
}
void Cleartree(NODEPTR proot) {
    if(proot != NULL) {
        Cleartree(proot -> left);
        Cleartree(proot -> right);
        Freenode(proot);
    }
}
void main(void) {
    NODEPTR ptree, p;
    int noidung, chucnang;
    Initialize(&ptree);
    do {
        clrscr();
        printf("\n CÂY HOÀN TOÀN CÂN BẰNG");
        printf("\n 1 - Thêm node trên cây");
        printf("\n 2 - Duyệt cây theo NLR");
        printf("\n 3 - Duyệt cây theo LNR");
```

```
printf("\n 4 - Duyệt cây theo LRN");
printf("\n 5 - Tìm kiếm trên cây");
printf("\n 6 - Loại bỏ toàn bộ cây");
printf("\n 0 - Thoat khỏi chương trình");
printf("\n Lựa chọn chức năng:");
scanf("%d", &chucnang);
switch(chucnang) {
    case 1:
        printf("\n Nội dung node mới:");
        scanf("%d", &noidung);
        if(ptree = NULL)
            ptree = Makenode(noidung);
        else
            Insert(ptree, noidung);
        break;
    case 2:
        printf("\n Duyệt cây theo NLR");
        if(ptree == FALSE)
            printf("\n Cây rỗng");
        else
            Pretrav(ptree);
        break;
    case 3:
        printf("\n Duyệt cây theo LNR");
        if(ptree == FALSE)
            printf("\n Cây rỗng");
        else
            Intrav(ptree);
```

```
            break;
        case 4:
            printf("\n Duyệt cây theo NRN");
            if(ptree == FALSE)
                printf("\n Cây rỗng");
            else
                Postrav(ptree);
            break;
        case 5:
            printf("\n Nội dung cần tìm:");
            scanf("%d", &noidung);
            if(Search(ptree, noidung))
                printf("\n Tìm thấy");
            else
                printf("\n Không tìm thấy");
            break;
        case 6:
            Cleartree(ptree); break;
    }
    delay(1000);
  } while(chucnang != 0);
  Cleartree(ptree); ptree = NULL;
}
```

5.8. CÀI ĐẶT CÂY NHỊ PHÂN TÌM KIẾM BẰNG LINK LIST

Vì cây nhị phân tìm kiếm là một dạng đặc biệt của cây nên các thao tác như thiết lập cây, duyệt cây vẫn như cây nhị phân thông thường riêng, các thao tác tìm kiếm, thêm node và loại bỏ node có thể được thực hiện như sau:

Thao tác tìm kiếm node (Search): Giả sử ta cần tìm kiếm node có giá trị x trên cây nhị phân tìm kiếm, trước hết ta bắt đầu từ gốc:

- Nếu cây rỗng: phép tìm kiếm không thoả mãn;
- Nếu x trùng với khoá gốc: phép tìm kiếm thoả mãn;
  - Nếu x nhỏ hơn khoá gốc thì tìm sang cây bên trái;
  - Nếu x lớn hơn khoá gốc thì tìm sang cây bên phải;

```
NODEPTR Search(NODEPTR proot, int x) {
    NODEPTR p; p = proot;
    if(p != NULL) {
        if(x <p -> infor)
            Search(proot -> left, x);
        if(x >p -> infor)
            Search(proot -> right, x);
    }
    return(p);
}
```

Thao tác chèn thêm node (Insert): để thêm node x vào cây nhị phân tìm kiếm, ta thực hiện như sau:

- Nếu x trùng với gốc thì không thể thêm node
- Nếu x < gốc và chưa có lá con bên trái thì thực hiện thêm node vào nhánh bên trái.
- Nếu x > gốc và chưa có lá con bên phải thì thực hiện thêm node vào nhánh bên phải.

```
void Insert(NODEPTR proot, int x) {
    if(x = proot -> infor) {
        printf("\n Nội dung bị trùng");
        delay(2000); return;
```

```
        }
        else if(x<proot -> infor && proot -> left == NULL) {
            Setleft(proot, x); return;
        }
        else if(x>proot -> infor && proot -> right == NULL) {
            Setright(proot, x); return;
        }
        else if(x<proot -> infor)
            Insert(proot -> left, x);
        else Insert(proot -> right, x);
    }
```

Thao tác loại bỏ node (Remove): Việc xoá node trên cây nhị phân tìm kiếm khá phức tạp. Vì sau khi xoá node, chúng ta phải điều chỉnh lại cây để nó vẫn là cây nhị phân tìm kiếm. Khi xoá node trên cây nhị phân tìm kiếm thì node cần xoá bỏ có thể ở một trong 3 trường hợp sau:

Trường hợp 1: nếu node p cần xoá là node lá hoặc node gốc thì việc loại bỏ được thực hiện ngay.

Trường hợp 2: nếu node p cần xoá có một cây con thì ta phải lấy node con của node p thay thế cho p.

Trường hợp 3: node p cần xoá có hai cây con. Nếu node cần xoá ở phía cây con bên trái thì node bên trái nhất sẽ được chọn làm node thế mạng, nếu node cần xoá ở phía cây con bên phải thì node bên phải nhất sẽ được chọn làm node thế mạng. Thuật toán loại bỏ node trên cây nhị phân tìm kiếm được thể hiện như sau:

```
NODEPTR Remove(NODEPTR p) {
NODEPTR rp, f;
if(p = NULL) {
    printf("\n Node p không có thực");
```

```
        delay(2000); return(p);
    }
    if(p -> right == NULL)
        rp = p -> left;
    else {
        if(p -> left == NULL)
            rp = p -> right;
        else {
            f = p;
            rp = p -> right;
            while(rp -> left != NULL) {
                f = rp;
                rp = rp -> left;
            }
            if(f != p) {
                f -> left = rp -> right;
                rp -> right = p -> right;
                rp -> left = p -> left;
            }
            else
                rp -> left = p -> left;
        }
    }
    Freenode(p);
    return(rp);
    }
```

Cài đặt các thao tác trên cây nhị phân tìm kiếm:

```
#include <stdio.h>
#include <stdlib.h>
```

```
#include <conio.h>
#include <alloc.h>
#include <string.h>
#include <dos.h>
#define TRUE 1
#define FALSE 0
#define MAX 100
struct node {
    int infor;
    struct node *left;
    struct node *right;
};
typedef struct node *NODEPTR;
NODEPTR Getnode(void) {
    NODEPTR p;
    p = (NODEPTR)malloc(sizeof(struct node));
    return(p);
}
void Freenode(NODEPTR p) {
    free(p);
}
void Initialize(NODEPTR *ptree) {
    *ptree = NULL;
}
NODEPTR Makenode(int x) {
    NODEPTR p;
    p = Getnode();
    p -> infor = x;
    p -> left = NULL;
    p -> right = NULL;
```

```
    return(p);
}
void Setleft(NODEPTR p, int x) {
    if(p == NULL)
        printf("\n Node p không có thực");
    else {
        if(p -> left != NULL)
            printf("\n Node con bên trái đã tồn tại");
        else
            p -> left = Makenode(x);
    }
}
void Setright(NODEPTR p, int x) {
    if(p == NULL)
        printf("\n Node p không có thực");
    else {
        if(p -> right != NULL)
            printf("\n Node con bên phải đã tồn tại");
        else
            p -> right = Makenode(x);
    }
}
void Pretrav(NODEPTR proot) {
    if(proot != NULL) {
        printf("%5d", proot -> infor);
        Pretrav(proot -> left);
        Pretrav(proot -> right);
    }
}
void Intrav(NODEPTR proot) {
```

```
    if(proot != NULL) {
        Intrav(proot -> left);
        printf("%5d", proot -> infor);
        Intrav(proot -> right);
    }
}
void Postrav(NODEPTR proot) {
    if(proot != NULL) {
        Postrav(proot -> left);
        Postrav(proot -> right);
        printf("%5d", proot -> infor);
    }
}
void Insert(NODEPTR proot, int x) {
    if(x = proot -> infor) {
        printf("\n Nội dung bị trùng");
        delay(2000); return;
    }
    else if(x<proot -> infor && proot -> left == NULL) {
        Setleft(proot, x); return;
    }
    else if(x>proot -> infor && proot -> right == NULL) {
        Setright(proot, x); return;
    }
    else if(x<proot -> infor)
        Insert(proot -> left, x);
    else Insert(proot -> right, x);
}
NODEPTR Search(NODEPTR proot, int x) {
    NODEPTR p; p = proot;
```

```
    if(p != NULL) {
        if(x < proot -> infor)
            p = Search(proot -> left, x);
        else if(x > proot -> infor)
            p = Search(proot -> right, x);
 }
 return(p);
}
NODEPTR Remove(NODEPTR p) {
    NODEPTR rp, f;
    if(p == NULL) {
        printf("\n Node p không có thực");
        delay(2000); return(p);
    }
    if(p -> right == NULL)
        rp = p -> left;
    else {
        if(p -> left == NULL)
            rp = p -> right;
        else {
            f = p;
            rp = p -> right;
            while(rp -> left != NULL) {
                f = rp;
                rp = rp -> left;
            }
            if(f != p) {
                f -> left = rp -> right;
                rp -> right = p -> right;
                rp -> left = p -> left;
```

```
            }
            else
                rp -> left = p -> left;
        }
    }
    Freenode(p);
    return(rp);
}
void Cleartree(NODEPTR proot) {
    if(proot != NULL) {
        Cleartree(proot -> left);
        Cleartree(proot -> right);
        Freenode(proot);
    }
}
void main(void) {
    NODEPTR ptree, p;
    int noidung, chucnang;
    Initialize(&ptree);
    do {
        clrscr();
        printf("\n CÂY NHỊ PHÂN TÌM KIẾM");
        printf("\n 1 - Thêm node trên cây");
        printf("\n 2 - Xóa node gốc");
        printf("\n 3 - Xóa node con bên trái");
        printf("\n 4 - Xóa node con bên phải");
        printf("\n 5 - Xóa toàn bộ cây");
        printf("\n 6 - Duyệt cây theo NLR");
        printf("\n 7 - Duyệt cây theo LNR");
        printf("\n 8 - Duyệt cây theo LRN");
```

```
printf("\n 9 - Tìm kiếm trên cây");
printf("\n 0 - Thoát khỏi chương trình");
printf("\n Lựa chọn chức năng:");
scanf("%d", &chucnang);
switch(chucnang) {
    case 1:
        printf("\n Nội dung node mới:");
        scanf("%d", &noidung);
        if(ptree == NULL)
            ptree = Makenode(noidung);
        else
            Insert(ptree, noidung);
        break;
    case 2:
        if(ptree == NULL)
            printf("\n Cây bị rỗng");
        else
            ptree = Remove(ptree);
        break;
    case 3:
        printf("\n Nội dung node cha:");
        scanf("%d", &noidung);
        p = Search(ptree, noidung);
        if(p != NULL)
            p -> left = Remove(p -> left);
        else
            printf("\n Không có node cha");
        break;
    case 4:
        printf("\n Nội dung node cha:");
```

```
        scanf("%d", &noidung);
        p = Search(ptree, noidung);
        if(p != NULL)
            p -> right = Remove(p -> right);
        else
            printf("\n Không có node cha");
        break;
    case 5:
        Cleartree(ptree);
        break;
    case 6:
        printf("\n Duyệt cây theo NLR");
        if(ptree == NULL)
            printf("\n Cây rỗng");
        else
            Pretrav(ptree);
        break;
    case 7:
        printf("\n Duyệt cây theo LNR");
        if(ptree == NULL)
            printf("\n Cây rỗng");
        else
            Intrav(ptree);
        break;
    case 8:
        printf("\n Duyệt cây theo NRN");
        if(ptree == NULL)
            printf("\n Cây rỗng");
        else
            Postrav(ptree);
```

```
            break;
        case 9:
            printf("\n Nội dung cần tìm:");
            scanf("%d", &noidung);
            if(Search(ptree, noidung))
                printf("\n Tìm thấy");
            else
                printf("\n Không tìm thấy");
            break;
    }
    delay(1000);
  } while(chucnang != 0);
  Cleartree(ptree); ptree = NULL;
}
```

BÀI TẬP CHƯƠNG 5

5.1. Một cây nhị phân được gọi là cây nhị phân đúng nếu node gốc của cây và các node trung gian đều có hai node con (ngoại trừ node lá). Chứng minh rằng, nếu cây nhị phân đúng có n node lá thì cây này có tất cả 2n-1 node. Hãy tạo lập một cây nhị phân bất kỳ, sau đó kiểm tra xem nếu cây không phải là cây nhị phân đúng hãy tìm cách bổ sung vào một số node để cây trở thành cây hoàn toàn đúng. Làm tương tự như trên với thao tác loại bỏ node.

5.2. Một cây nhị phân được gọi là cây nhị phân đầy với chiều sâu d (d nguyên dương) khi và chỉ khi ở mức i ($0 \le i \le d$) cây có đúng 2i node. Hãy viết chương trình kiểm tra xem một cây nhị phân có phải là một cây đầy hay không? Nếu cây chưa phải là cây nhị phân đầy, hãy tìm cách bổ sung một số node vào cây nhị phân để nó trở thành cây nhị phân đầy.

5.3. Một cây nhị phân được gọi là cây nhị phân gần đầy với độ sâu d nếu với mọi mức i ($0 \le i \le d - 1$) nó có đúng 2^i node. Cho cây nhị phân bất kỳ, hãy kiểm tra xem nó có phải là cây nhị phân gần đầy hay không?

5.4. Hãy xây dựng các thao tác sau trên cây nhị phân:

- Tạo lập cây nhị phân;
- Đếm số node của cây nhị phân;
- Xác định chiều sâu của cây nhị phân;
- Xác định số node lá của cây nhị phân;
- Xác định số node trung gian của cây nhị phân;
- Xác định số node trong từng mức của cây nhị phân;

- Xây dựng tập thao tác tương tự như trên đối với các nhánh cây con;
- Thêm một node vào node phải của một node;
- Thêm node vào node trái của một node;
- Loại bỏ node phải của một node;
- Loại bỏ node trái của một node;
- Loại bỏ cả cây;
- Duyệt cây theo thứ tự trước;
- Duyệt cây theo thứ giữa;
- Duyệt cây theo thứ tự sau;

5.5. Cho file dữ liệu cay.in được tổ chức thành từng dòng, trên mỗi dòng ghi lại một từ là nội dung node của một cây nhị phân tìm kiếm. Hãy xây dựng các thao tác sau cho cây nhị phân tìm kiếm:

Tạo lập cây nhị phân tìm kiếm với node gốc là từ đầu tiên trong file dữ liệu cay.in.

- Xác định số node trên cây nhị phân tìm kiếm;
- Xác định chiều sâu của cây nhị phân tìm kiếm;
- Xác định số node nhánh cây bên trái;
- Xác định số node nhánh cây con bên phải;
- Xác định số node trung gian;
- Xác định số node lá;
- Tìm node có độ dài lớn nhất;
- Thêm node;
- Loại bỏ node;
- Loại bỏ cả cây;

- Duyệt cây theo thứ tự trước;
- Duyệt cây theo thứ tự giữa;
- Duyệt cây theo thứ tự sau;

5.6. Cho cây nhị phân bất kỳ hãy xây dựng chương trình xác định xem:

- Cây có phải là cây nhị phân đúng hay không?
- Cây có phải là cây nhị phân đầy hay không ?
- Cây có phải là cây nhị phân gần đầy hay không?
- Cây có phải là cây nhị phân hoàn toàn cân bằng hay không?
- Cây có phải là cây nhị phân tìm kiếm hay không ?

5.7. Cho tam giác số được biểu diễn như hình dưới đây. Hãy viết chương trình tìm dãy các số có tổng lớn nhất trên con đường từ đỉnh và kết thúc tại đâu đó ở đáy. Biết rằng, mỗi bước đi có thể đi chéo xuống phía trái hoặc chéo xuống phía phải. Số lượng hàng trong tam giác là lớn hơn 1 nhưng nhỏ hơn 100; các số trong tam giác đều là các số từ 0 ... 99.

```
          7
        3   8
      8   1   0
    2   7   4   4
  4   5   2   6   5
```

Dữ liệu vào cho bởi file cay.in, dòng đầu tiên ghi lại số tự nhiên n là số lượng hàng trong tam giác, n hàng tiếp theo ghi lại từng hàng mỗi phần tử được phân biệt với nhau bởi một hoặc vài dấu trống. Kết quả ghi lại trong file cay.out dòng đầu tiên ghi lại tổng số lớn nhất tìm được, dòng kế tiếp ghi lại dãy các số có tổng lớn nhất. Ví dụ với hình trên file input và output như sau:

cay.in

5

7

8

8 1 0

2 7 4 4

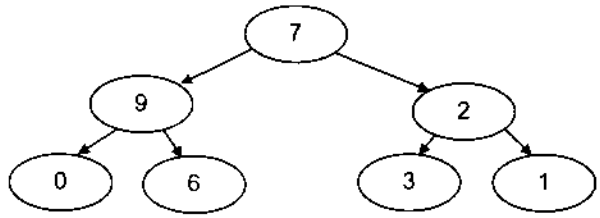
4 5 2 6 5

cay.out

30

7 3 8 7 5

5.8. Cho cây nhị phân số hoàn toàn cân bằng: (số node bên nhánh cây con bên trái đúng bằng số node nhánh cây con bên phải, ở mức thứ i có đúng 2^i node) như hình sau:



Hãy tìm dãy các node xuất phát từ gốc tới một node lá nào đó sao cho tổng giá trị của các node là lớn nhất, biết rằng mỗi bước đi chỉ được phép đi chéo sang node trái hoặc chéo theo node phải. Dữ liệu vào cho bởi file cay.in, dòng đầu tiên ghi lại số tự nhiên $n \leq 50$ là số các mức của cây, n dòng kế tiếp mỗi dòng ghi lại dãy các số là các node trên mỗi mức. Kết quả ghi lại trong file cay.out theo thứ tự, dòng đầu là tổng lớn nhất của hành trình, dòng kế tiếp là dãy các node trong hành trình. Ví dụ: với hình trên file input & output được tổ chức như sau:

cay.in

3

7

9 2

0 6 3 1

cay.out

22

7 9 6

CHƯƠNG 6

ĐỒ THỊ (GRAPH)

6.1. NHỮNG KHÁI NIỆM CƠ BẢN VỀ ĐỒ THỊ

6.1.1. Các loại đồ thị

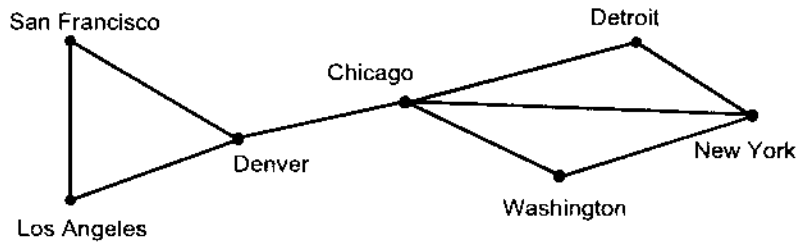
Lý thuyết đồ thị là lĩnh vực nghiên cứu đã tồn tại từ những năm đầu của thế kỷ 18 nhưng lại có những ứng dụng hiện đại. Những tư tưởng cơ bản của lý thuyết đồ thị được nhà toán học người Thuỵ Sĩ Leonhard Euler đề xuất và chính ông là người dùng lý thuyết đồ thị giải quyết bài toán nổi tiếng "Cầu Konigsberg".

Đồ thị được sử dụng để giải quyết nhiều bài toán thuộc các lĩnh vực khác nhau. Chẳng hạn, ta có thể dùng đồ thị để biểu diễn những mạch vòng của một mạch điện, dùng đồ thị biểu diễn quá trình tương tác giữa các loài trong thế giới động thực vật, dùng đồ thị biểu diễn những đồng phân của các hợp chất polyme hoặc biểu diễn mối liên hệ giữa các loại thông tin khác nhau. Có thể nói, lý thuyết đồ thị được ứng dụng rộng rãi trong tất cả các lĩnh vực khác nhau của thực tế cũng như những lĩnh vực trừu tượng của lý thuyết tính toán.

Đồ thị (Graph) là một cấu trúc dữ liệu rời rạc bao gồm các đỉnh và các cạnh nối các cặp đỉnh này. Chúng ta phân biệt đồ thị thông qua kiểu và số lượng cạnh nối giữa các cặp đỉnh của đồ thị. Để minh chứng cho các loại đồ thị, chúng ta xem xét một số ví dụ về các loại mạng máy tính bao gồm: mỗi máy tính là một đỉnh, mỗi cạnh là những kênh điện thoại được nối giữa hai máy tính với nhau. Hình 6.1 là sơ đồ của mạng máy tính đơn kênh thoại.

Trong mạng máy tính này, mỗi máy tính là một đỉnh của đồ thị, mỗi cạnh vô hướng biểu diễn các đỉnh nối hai đỉnh phân biệt, không

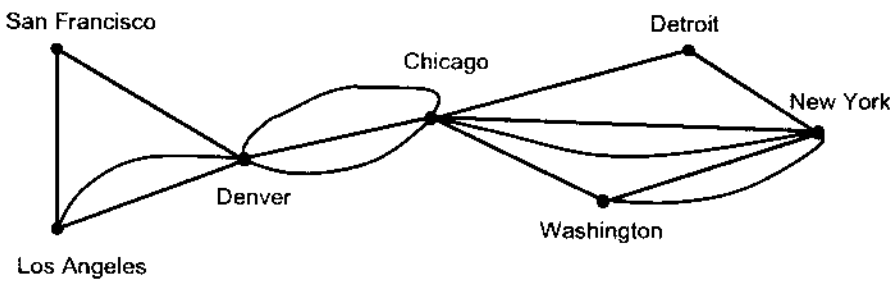
có hai cặp đỉnh nào nối cùng một cặp đỉnh. Mạng loại này có thể biểu diễn bằng một **đơn đồ thị vô hướng**.



Hình 6.1. Mạng máy tính đơn kênh thoại.

Định nghĩa 1. *Đơn đồ thị vô hướng G = <V, E> bao gồm V là tập các đỉnh, E là tập các cặp có thứ tự gồm hai phần tử khác nhau của V gọi là các cạnh.*

Trong trường hợp giữa hai máy tính nào đó thường xuyên truyền tải nhiều thông tin, người ta nối hai máy tính bởi nhiều kênh thoại khác nhau. Mạng máy tính đa kênh thoại có thể được biểu diễn như hình 6.2.



Hình 6.2. Mạng máy tính đa kênh thoại

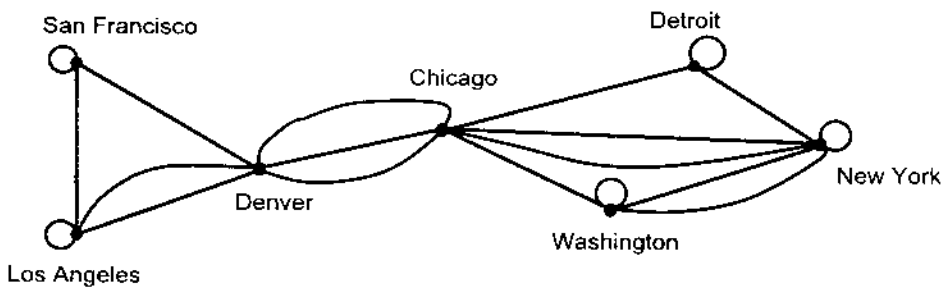
Trên hình 6.2, giữa hai máy tính có thể được nối với nhau bởi nhiều hơn một kênh thoại. Với mạng loại này, chúng ta không thể dùng đơn đồ thị vô hướng để biểu diễn. Đồ thị loại này là đa đồ thị vô hướng.

Định nghĩa 2. *Đa đồ thị vô hướng G = <V, E> bao gồm V là tập các đỉnh, E là họ các cặp không có thứ tự gồm hai phần tử khác nhau*

của V gọi là tập các cạnh. e_1, e_2 được gọi là cạnh lặp nếu chúng cùng tương ứng với một cặp đỉnh.

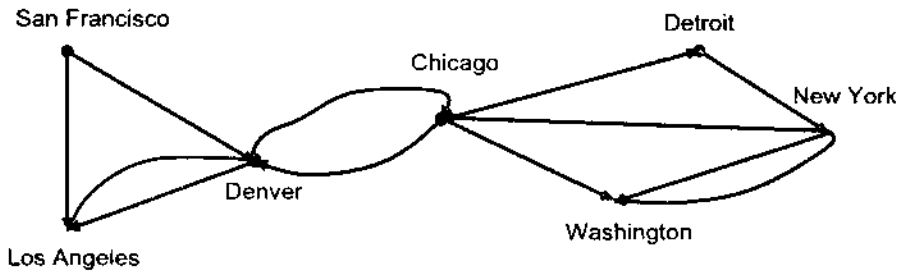
Rõ ràng, mọi đơn đồ thị đều là đa đồ thị, nhưng không phải đa đồ thị nào cũng là đơn đồ thị vì giữa hai đỉnh có thể có nhiều hơn một cạnh nối giữa chúng với nhau. Trong nhiều trường hợp, có máy tính có thể nối nhiều kênh thoại với chính nó. Với loại mạng này, ta không thể dùng đa đồ thị để biểu diễn mà phải dùng giả đồ thị vô hướng. Giả đồ thị vô hướng được mô tả như trong hình 6.3.

Định nghĩa 3. *Giả đồ thị vô hướng G = <V, E> bao gồm V là tập đỉnh, E là họ các cặp không có thứ tự gồm hai phần tử (hai phần tử không nhất thiết phải khác nhau) trong V được gọi là các cạnh. Cạnh e được gọi là khuyên nếu có dạng e = (u, u), trong đó u là đỉnh nào đó thuộc V.*



Hình 6.3. Mạng máy tính đa kênh thoại có khuyên

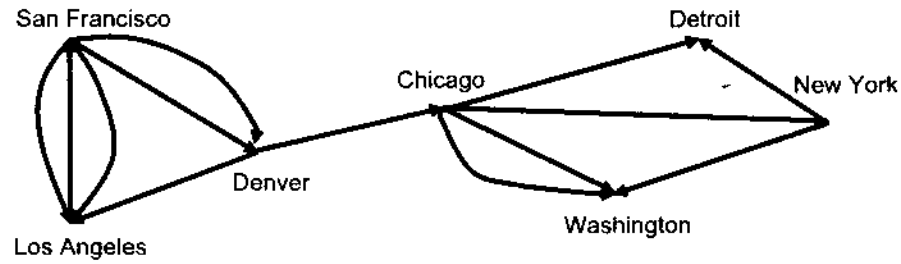
Trong nhiều mạng, các kênh thoại nối giữa hai máy tính có thể chỉ được phép truyền tin theo một chiều. Chẳng hạn máy tính đặt tại San Francisco được phép truy nhập tới máy tính đặt tại Los Angeles, nhưng máy tính đặt tại Los Angeles không được phép truy nhập ngược lại San Francisco. Hoặc máy tính đặt tại Denver có thể truy nhập được tới máy tính đặt tại Chicago và ngược lại máy tính đặt tại Chicago cũng có thể truy nhập ngược lại máy tính tại Denver. Để mô tả mạng loại này, chúng ta dùng khái niệm đơn đồ thị có hướng. Đơn đồ thị có hướng được mô tả như trong hình 6.4.



Hình 6.4. Mạng máy tính có hướng.

Định nghĩa 4. *Đơn đồ thị có hướng G = <V, E> bao gồm V là tập các đỉnh, E là tập các cặp có thứ tự gồm hai phần tử của V gọi là các cung.*

Đồ thị có hướng trong hình 6.4 không chứa các cạnh bội. Nên đối với các mạng đa kênh thoại một chiều, đồ thị có hướng không thể mô tả được mà ta dùng khái niệm đa đồ thị có hướng. Mạng có dạng đa đồ thị có hướng được mô tả như trong hình 6.5.



Hình 6.5. Mạng máy tính đa kênh thoại một chiều.

Định nghĩa 5. *Đa đồ thị có hướng G = <V, E> bao gồm V là tập đỉnh, E là cặp có thứ tự gồm hai phần tử của V được gọi là các cung. Hai cung e_1, e_2 tương ứng với cùng một cặp đỉnh được gọi là cung lặp.*

Từ những dạng khác nhau của đồ thị kể trên, chúng ta thấy sự khác nhau giữa các loại đồ thị được phân biệt thông qua các cạnh của đồ thị có thứ tự hay không có thứ tự, các cạnh bội, khuyên có được dùng hay không. Ta có thể tổng kết các loại đồ thị thông qua bảng 6.1.

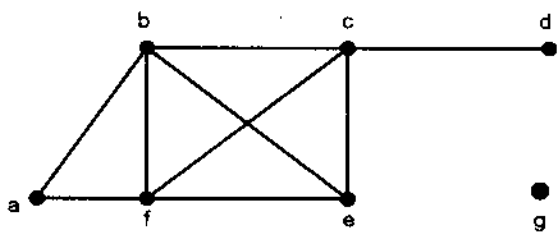
Bảng 6.1. Phân biệt các loại đồ thị

Loại đồ thị	Cạnh	Có cạnh bội	Có khuyên
Đơn đồ thị vô hướng	Vô hướng	Không	Không
Đa đồ thị vô hướng	Vô hướng	Có	Không
Giả đồ thị vô hướng	Vô hướng	Có	Có
Đồ thị có hướng	Có hướng	Không	Có
Đa đồ thị có hướng	Có hướng	Có	Có

6.1.2. Các thuật ngữ cơ bản

Định nghĩa 1. *Hai đỉnh u và v của đồ thị vô hướng G = <V, E> được gọi là kề nhau nếu (u, v) là cạnh thuộc đồ thị G. Nếu e = (u, v) là cạnh của đồ thị G thì ta nói cạnh này liên thuộc với hai đỉnh u và v, hoặc ta nói cạnh e nối đỉnh u với đỉnh v, đồng thời các đỉnh u và v sẽ được gọi là đỉnh đầu của cạnh (u, v).*

Định nghĩa 2. *Ta gọi bậc của đỉnh v trong đồ thị vô hướng là số cạnh liên thuộc với nó và ký hiệu là deg(v).*



Hình 6.6 Đồ thị vô hướng G

Ví dụ 1. Xét đồ thị trong hình 6.6, ta có:

deg(a) = 2, deg(b) = deg(c) = deg(f) = 4, deg(e) = 3, deg(d) = 1, deg(g) = 0.

Đỉnh bậc 0 được gọi là đỉnh cô lập. Đỉnh bậc 1 được gọi là đỉnh treo. Trong ví dụ trên, đỉnh g là đỉnh cô lập, đỉnh d là đỉnh treo.

Định lý 1. *Giả sử G = <V, E> là đồ thị vô hướng với m cạnh. Khi đó* $2m = \sum_{v \in V} deg(v)$.

Chứng minh: Rõ ràng mỗi cạnh e = (u, v) bất kỳ, được tính một lần trong deg(u) và một lần trong deg(v). Từ đó suy ra số tổng tất cả các bậc bằng hai lần số cạnh.

Hệ quả. Trong đồ thị vô hướng G = <V, E>, số các đỉnh bậc lẻ là một số chẵn.

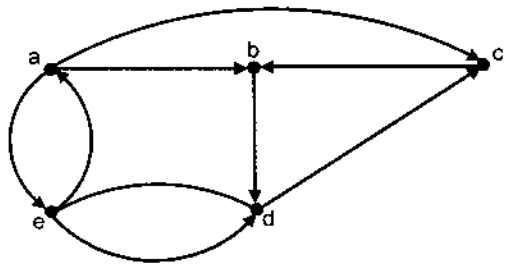
Chứng minh: Gọi O là tập các đỉnh bậc chẵn và V là tập các đỉnh bậc lẻ. Từ định lý 1 ta suy ra:

$$2m = \sum_{v \in V} \deg(v) = \sum_{v \in O} \deg(v) + \sum_{v \in U} \deg(v)$$

Do deg(v) là chẵn với v là đỉnh trong O nên tổng thứ hai trong vế phải cũng là một số chẵn.

Định nghĩa 3. *Nếu e = (u, v) là cung của đồ thị có hướng G thì ta nói hai đỉnh u và v là kề nhau, và nói cung (u, v) nối đỉnh u với đỉnh v hoặc cũng nói cung này đi ra khỏi đỉnh u và đi vào đỉnh v. Đỉnh u (v) sẽ được gọi là đỉnh đầu (cuối) của cung (u, v).*

Định nghĩa 4. *Ta gọi bán bậc ra (bán bậc vào) của đỉnh v trong đồ thị có hướng là số cung của đồ thị đi ra khỏi nó (đi vào nó) và ký hiệu là $deg^+(v)$ và $deg^-(v)$.*



Hình 6.7. Đồ thị có hướng G.

Ví dụ 2. Xét đồ thị có hướng trong hình 6.7, ta có:

$deg^-(a) = 1$, $deg^-(b) = 2$, $deg^-(c) = 2$, $deg^-(d) = 2$, $deg^-(e) = 2$.

$deg^+(a) = 3$, $deg^+(b) = 1$, $deg^+(c) = 1$, $deg^+(d) = 2$, $deg^+(e) = 2$.

Do mỗi cung (u, v) được tính một lần trong bán bậc vào của đỉnh v và một lần trong bán bậc ra của đỉnh u nên ta có:

Định lý 2. *Giả sử G = <V, E> là đồ thị có hướng. Khi đó* $\sum_{v \in V} deg^{+}(v) = \sum_{v \in V} deg^{-}(v) = / E /$

Rất nhiều tính chất của đồ thị có hướng không phụ thuộc vào hướng trên các cung của nó. Vì vậy, trong nhiều trường hợp, ta bỏ qua các hướng trên cung của đồ thị. Đồ thị vô hướng nhận được bằng cách bỏ qua hướng trên các cung được gọi là đồ thị vô hướng tương ứng với đồ thị có hướng đã cho.

6.1.3 Đường đi, chu trình, đồ thị liên thông

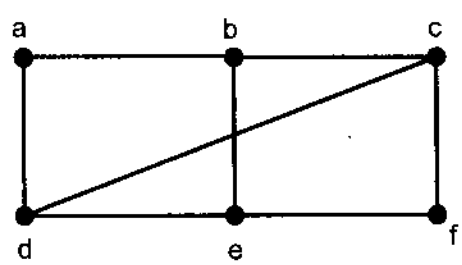
Định nghĩa 1. *Đường đi độ dài n từ đỉnh u đến đỉnh v trên đồ thị vô hướng G = <V, E> là dãy* $x_0, x_1, \ldots, x_{n-1}, x_n$.

Trong đó n là số nguyên dương, $x_0 = u$, $x_n = v$, $(x_i, x_{i+1}) \in E$, $i = 0, 1, 2, \ldots, n-1$

Đường đi như trên còn có thể biểu diễn thành dãy các cạnh:

$(x_0, x_1), (x_1, x_2), \ldots, (x_{n-1}, x_n)$.

Đỉnh u là đỉnh đầu, đỉnh v là đỉnh cuối của đường đi. Đường đi có đỉnh đầu trùng với đỉnh cuối (u = v) được gọi là chu trình. Đường đi hay chu trình được gọi là đơn nếu như không có cạnh nào lặp lại.



Hình 6.8. Đường đi trên đồ thị

Ví dụ 3. Tìm các đường đi, chu trình trong đồ thị vô hướng như trong hình 6.8.

a, d, c, f, e là đường đi đơn độ dài 4, d, e, c, a không là đường đi vì (e, c) không phải là cạnh của đồ thị. Dãy b, c, f, e, b là chu trình độ dài 4. Đường đi a, b, e, d, a, b có độ dài 5 không phải là đường đi đơn vì cạnh (a, b) có mặt hai lần.

Khái niệm đường đi và chu trình trên đồ thị có hướng được định nghĩa hoàn toàn tương tự, chỉ có điều khác biệt duy nhất là ta phải chú ý tới các cung của đồ thị.

Định nghĩa 2. *Đường đi độ dài n từ đỉnh u đến đỉnh v trong đồ thị có hướng* $G = <V, A>$ *là dãy* $x_0, x_1, \ldots, x_n$.

Trong đó, n là số nguyên dương, $u = x_0$, $v = x_n$, $(x_i, x_{i+1}) \in A$.

Đường đi như trên có thể biểu diễn thành dãy các cung:

$$(x_0, x_1), (x_1, x_2), \ldots, (x_{n-1}, x_n).$$

Đỉnh u được gọi là đỉnh đầu, đỉnh v được gọi là đỉnh cuối của đường đi. Đường đi có đỉnh đầu trùng với đỉnh cuối (u = v) được gọi là một chu trình. Đường đi hay chu trình được gọi là đơn nếu như không có hai cạnh nào lặp lại.

Định nghĩa 3. *Đồ thị vô hướng được gọi là liên thông nếu luôn tìm được đường đi giữa hai đỉnh bất kỳ của nó.*

6.2. BIỂU DIỄN ĐỒ THỊ TRÊN MÁY TÍNH

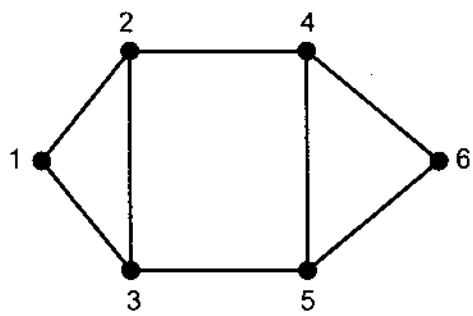
6.2.1. Ma trận kề, ma trận trọng số

Để lưu trữ đồ thị và thực hiện các thuật toán khác nhau, ta cần phải biểu diễn đồ thị trên máy tính, đồng thời sử dụng những cấu trúc dữ liệu thích hợp để mô tả đồ thị. Việc chọn cấu trúc dữ liệu nào để biểu diễn đồ thị có tác động rất lớn đến hiệu quả thuật toán. Vì vậy, lựa chọn cấu trúc dữ liệu thích hợp biểu diễn đồ thị sẽ phụ thuộc vào từng bài toán cụ thể.

Xét đồ thị đơn vô hướng G = <V, E>, với tập đỉnh V = {1, 2,..., n}, tập cạnh E = {e_1, e_2,..., e_m}. Ta gọi ma trận kề của đồ thị G là ma trận có các phần tử hoặc bằng 0 hoặc bằng 1 theo qui định như sau:

A = { a_{ij}: a_{ij} = 1 nếu (i, j) ∈E, a_{ij} = 0 nếu (i, j) ∉E; i, j = 1, 2,..., n}.

Ví dụ 1. Biểu diễn đồ thị trong hình 6.9 dưới đây bằng ma trận kề.



$$\begin{array}{c|cccccc} & 1 & 2 & 3 & 4 & 5 & 6 \\ \hline 1 & 0 & 1 & 1 & 0 & 0 & 0 \\ 2 & 1 & 0 & 1 & 1 & 0 & 0 \\ 3 & 1 & 1 & 0 & 0 & 1 & 0 \\ 4 & 0 & 1 & 0 & 0 & 1 & 1 \\ 5 & 0 & 0 & 1 & 1 & 0 & 1 \\ 6 & 0 & 0 & 0 & 1 & 1 & 0 \end{array}$$

Hình 6.9. Đồ thị vô hướng G

Tính chất của ma trận kề:

a) Ma trận kề của đồ thị vô hướng là ma trận đối xứng A[i, j] = A[j, i]; i, j = 1, 2,... n. Ngược lại, mỗi [0, 1] ma trận cấp n đẳng cấu với một đơn đồ thị vô hướng n đỉnh;

b) Tổng các phần tử theo dòng i (cột j) của ma trận kề chính bằng bậc đỉnh i (đỉnh j);

c) Nếu ký hiệu:

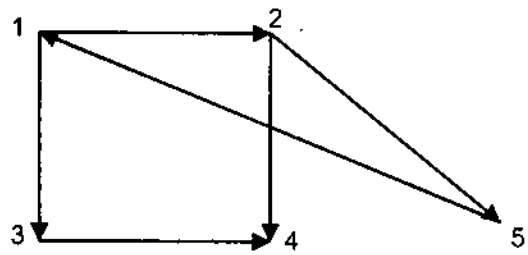
$a_{ij}^p, i, j = 1, 2, ..., n$ là các phần tử của ma trận

A^p = A.A... A (p lần) khi đó $a_{ij}^p, i, j = 1, 2, ..., n$,

Cho ta số đường đi khác nhau từ đỉnh i đến đỉnh j qua p-1 đỉnh trung gian.

Ma trận kề của đồ thị có hướng cũng được định nghĩa hoàn toàn tương tự, chúng ta chỉ cần lưu ý tới hướng của cạnh. Ma trận kề của đồ thị có hướng là không đối xứng.

Ví dụ 2. Tìm ma trận kề của đồ thị có hướng trong hình 6.10.

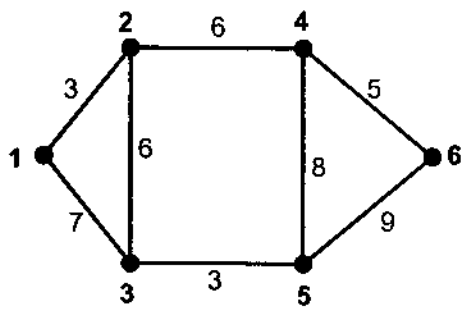


$$\begin{array}{c|ccccc} & 1 & 2 & 3 & 4 & 5 \\ \hline 1 & 0 & 1 & 1 & 0 & 0 \\ 2 & 0 & 0 & 0 & 1 & 1 \\ 3 & 0 & 0 & 0 & 1 & 0 \\ 4 & 0 & 0 & 0 & 0 & 0 \\ 5 & 1 & 0 & 0 & 0 & 0 \end{array}$$

Hình 6.10. Đồ thị có hướng G

Trong rất nhiều ứng dụng khác nhau của lý thuyết đồ thị, mỗi cạnh e = (u, v) của nó được gán bởi một số c(e) = c(u, v) gọi là trọng số của cạnh e. Đồ thị trong trường hợp như vậy gọi là đồ thị trọng số. Trong trường hợp đó, ma trận kề của đồ thị được thay bởi ma trận trọng số c = c[i, j], i, j = 1, 2,..., n. c[i, j] = c(i, j) nếu (i, j) $\in$E, c[i, j] = θ nếu (i, j) $\notin$E. Trong đó, θ nhận các giá trị: 0, ∞, $-\infty$ tuỳ theo từng tình huống cụ thể của thuật toán.

Ví dụ 3. Ma trận kề của đồ thị có trọng số trong hình 6.11.



$$\begin{array}{c|cccccc} & 1 & 2 & 3 & 4 & 5 & 6 \\ \hline 1 & 0 & 3 & 7 & 0 & 0 & 0 \\ 2 & 3 & 0 & 6 & 6 & 0 & 0 \\ 3 & 7 & 6 & 0 & 0 & 3 & 0 \\ 4 & 0 & 6 & 0 & 0 & 8 & 5 \\ 5 & 0 & 0 & 3 & 8 & 0 & 9 \\ 6 & 0 & 0 & 0 & 5 & 8 & 0 \end{array}$$

Hình 6.11. Đồ thị trọng số G

Ưu điểm của phương pháp biểu diễn đồ thị bằng ma trận kề (hoặc ma trận trọng số) là ta dễ dàng trả lời được câu hỏi: Hai đỉnh u, v có kề nhau trên đồ thị hay không và chúng ta chỉ mất đúng một phép so sánh. Nhược điểm lớn nhất của nó là bất kể đồ thị có bao nhiêu cạnh ta đều mất n^2 đơn vị bộ nhớ để lưu trữ đồ thị.

6.2.2. Danh sách cạnh (cung)

Trong trường hợp đồ thị thưa (đồ thị có số cạnh $m \leq 6n$), người ta thường biểu diễn đồ thị dưới dạng danh sách cạnh. Trong phép biểu diễn này, chúng ta sẽ lưu trữ danh sách tất cả các cạnh (cung) của đồ thị vô hướng (có hướng). Mỗi cạnh (cung) e (x, y) được tương ứng với hai biến dau[e], cuoi[e]. Như vậy, để lưu trữ đồ thị, ta cần 2m đơn vị bộ nhớ. Nhược điểm lớn nhất của phương pháp này là để nhận biết những cạnh nào kề với cạnh nào chúng ta cần m phép so sánh trong khi duyệt qua tất cả m cạnh (cung) của đồ thị. Nếu là đồ thị có trọng số, ta cần thêm m đơn vị bộ nhớ để lưu trữ trọng số của các cạnh.

Ví dụ 4. Danh sách cạnh (cung) của đồ thị vô hướng trong hình 6.9, đồ thị có hướng hình 6.10, đồ thị trọng số hình 6.11.

Dau	Cuoi	Dau	Cuoi	Dau	Cuoi	Trongso
1	2	1	2	1	2	3
1	3	1	3	1	3	7
2	3	2	4	2	3	6
2	4	2	5	2	4	6
3	5	3	4	3	5	3
4	5	5	1	4	5	8
4	6		4	4	6	5
5	6		5	5	6	9
Danh sách cạnh (cung) hình 6.9		Danh sách cạnh (cung) hình 6.10		Danh sách trọng số hình 6.11		

6. 2.3. Danh sách kề

Trong rất nhiều ứng dụng, cách biểu diễn đồ thị dưới dạng danh sách kề thường được sử dụng. Trong biểu diễn này, với mỗi đỉnh v của đồ thị chúng ta lưu trữ danh sách các đỉnh kề với nó mà ta ký hiệu là Ke(v), nghĩa là:

$$Ke(v) = \{u \in V: (u, v) \in E\},$$

Với cách biểu diễn này, mỗi đỉnh i của đồ thị, ta làm tương ứng với một danh sách tất cả các đỉnh kề với nó và được ký hiệu là List(i). Để biểu diễn List(i), ta có thể dùng các kiểu dữ liệu kiểu tập hợp, mảng hoặc danh sách liên kết.

Ví dụ 5. Danh sách kề của đồ thị vô hướng trong hình 6.9, đồ thị có hướng trong hình 6.10 được biểu diễn bằng danh sách kề như sau:

Đỉnh	List(i)
1	2 3
2	1 3 4
3	1 2 5
4	2 5 6
5	3 4 6
6	4 5

Đỉnh	List(i)
1	3 2
2	4 5
3	4
5	1

6.3. CÁC THUẬT TOÁN TÌM KIẾM TRÊN ĐỒ THỊ

6.3.1. Thuật toán tìm kiếm theo chiều sâu

Rất nhiều thuật toán trên đồ thị được xây dựng trên việc duyệt tất cả các đỉnh của đồ thị sao cho mỗi đỉnh được viếng thăm đúng một lần. Những thuật toán như vậy được gọi là thuật toán tìm kiếm trên đồ thị. Chúng ta cũng sẽ làm quen với hai thuật toán tìm kiếm cơ bản, đó là tìm kiếm theo chiều sâu (Depth First Search) và tìm kiếm theo chiều rộng (Breath First Search).

Tư tưởng cơ bản của thuật toán tìm kiếm theo chiều sâu là bắt đầu tại một đỉnh v_0 nào đó, chọn một đỉnh u bất kỳ kề với v_0 và lấy nó làm đỉnh duyệt tiếp theo. Cách duyệt tiếp theo được thực hiện tương tự như đối với đỉnh v_0.

Để kiểm tra việc duyệt mỗi đỉnh đúng một lần, chúng ta sử dụng một mảng gồm n phần tử (tương ứng với n đỉnh), nếu đỉnh thứ i đã được duyệt, phần tử tương ứng trong mảng có giá trị FALSE. Ngược lại, nếu đỉnh chưa được duyệt, phần tử tương ứng trong mảng có giá trị TRUE. Thuật toán tìm kiếm theo chiều sâu bắt đầu từ đỉnh v nào đó sẽ duyệt tất cả các đỉnh liên thông với v. Thuật toán có thể được mô tả bằng thủ tục đệ qui DFS () trong đó: chuaxet - là mảng các giá trị logic được thiết lập giá trị TRUE.

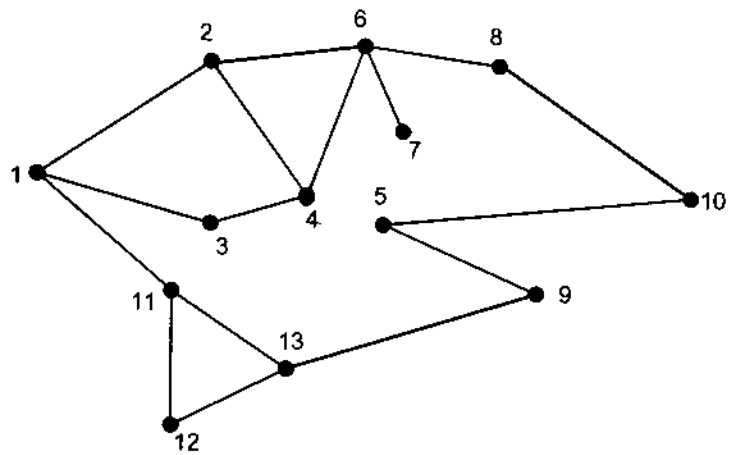
```
Procedure DFS(v);
begin
   Tham_Dinh(v); chuaxet[v] := FALSE;
   for u ∈ke(v) to n do
         if(chuaxet[u]) then
            DFS(A, n, v, chuaxet);
end;
```

Thủ tục DFS() sẽ thăm tất cả các đỉnh cùng thành phần liên thông với v mỗi đỉnh đúng một lần. Để đảm bảo duyệt tất cả các đỉnh của đồ thị (có thể có nhiều thành phần liên thông), chúng ta chỉ cần thực hiện:

```
begin
   for i:= 1 to n do
      chuaxet[i] := TRUE;
   for i:= 1 to n do
      if(chuaxet[i]) then
         DFS(A, n, i, chuaxet);
end;
```

Chú ý: Thuật toán tìm kiếm theo chiều sâu dễ dàng áp dụng cho đồ thị có hướng. Đối với đồ thị có hướng, chúng ta chỉ cần thay các cạnh vô hướng bằng các cung của đồ thị có hướng.

Ví dụ 1. Áp dụng thuật toán tìm kiếm theo chiều sâu với đồ thị trong hình sau:



Hình 6.12. Đồ thị vô hướng G

Kết quả duyệt: 1, 2, 4, 3, 6, 7, 8, 10, 5, 9, 13, 11, 12

Chương trình được thể hiện như sau:

```
#include <stdio.h>
#include <conio.h>
#include <io.h>
#include <stdlib.h>
#include <dos.h>
#define MAX    100
#define TRUE 1
#define FALSE 0
/* Tìm kiếm theo chiều sâu */
void Init(int A[][MAX], int *n) {
```

```
    FILE *fp; int i, j;
    fp = fopen("DFS.IN", "r");
    if(fp == NULL) {
        printf("\n Không có tệp input");
        delay(2000); return;
    }
    fscanf(fp, "%d", n);
    printf("\n Số đỉnh đồ thị:%d", *n);
    printf("\n Ma trận kề của đồ thị:");
    for(i = 1; i <= *n; i++) {
        printf("\n");
        for(j = 1; j <= *n; j++) {
            fscanf(fp, "%d", &A[i][j]);
            printf("%3d", A[i][j]);
        }
    }
}
void DFS(int A[][MAX], int n, int v, int chuaxet[]) {
    int u;
    printf("%3d", v); chuaxet[v] = FALSE;
    for(u = 1; u <= n; u++) {
        if(A[v][u] == 1 && chuaxet[u])
            DFS(A, n, u, chuaxet);
    }
}
void main(void) {
    int A[MAX][MAX], n, chuaxet[MAX];
    Init(A, &n);
```

```
    for(int i = 1; i <= n; i++)
        chuaxet[i] = TRUE;
    printf("\n\n");
    for(i = 1; i <= n; i++)
        if(chuaxet[i])
            DFS(n);
    getch();
}
```

6.3.2. Thuật toán tìm kiếm theo chiều rộng (Breadth First Search)

Để ý rằng, với thuật toán tìm kiếm theo chiều sâu, đỉnh thăm càng muộn sẽ trở thành đỉnh sớm được duyệt xong. Đó là kết quả tất yếu vì các đỉnh thăm được nạp vào stack trong thủ tục đệ qui. Khác với thuật toán tìm kiếm theo chiều sâu, thuật toán tìm kiếm theo chiều rộng thay thế việc sử dụng stack bằng hàng đợi queue. Trong thủ tục này, đỉnh được nạp vào hàng đợi đầu tiên là v, các đỉnh kề với v (v_1, v_2,..., v_k) được nạp vào queue kế tiếp. Quá trình được thực hiện tương tự với các đỉnh trong hàng đợi. Thuật toán dừng khi ta đã duyệt hết các đỉnh kề với đỉnh trong hàng đợi. Chúng ta có thể mô tả thuật toán bằng thủ tục BFS.

chuaxet - mảng kiểm tra các đỉnh đã xét hay chưa;

queue - hàng đợi lưu trữ các đỉnh sẽ được duyệt của đồ thị;

```
procedure BFS(u);
begin
    queue := ϕ;
    u <= queue;                    (* nạp u vào hàng đợi *)
    chuaxet[u] := false;
    while(queue ≠ ϕ) do
    begin
        queue <= p;                (* lấy p ra từ stack *)
```

```
            Tham_Dinh(p);
            for v ∈ ke(p) do
                if(chuaxet[v]) then
                begin
                    v <= queue;                (* nạp v vào hàng đợi *)
                    chuaxet[v] := false;
                end;
        end;
    end;
```

Thủ tục BFS sẽ thăm tất cả các đỉnh dùng thành phần liên thông với u. Để thăm tất cả các đỉnh của đồ thị, chúng ta chỉ cần thực hiện gọi tới thủ tục BFS().

```
begin
    for u ∈ V to n do
        chuaxet[u] := TRUE;
    for u ∈V do
        if(chuaxet[u] then
            BFS(n);
end;
```

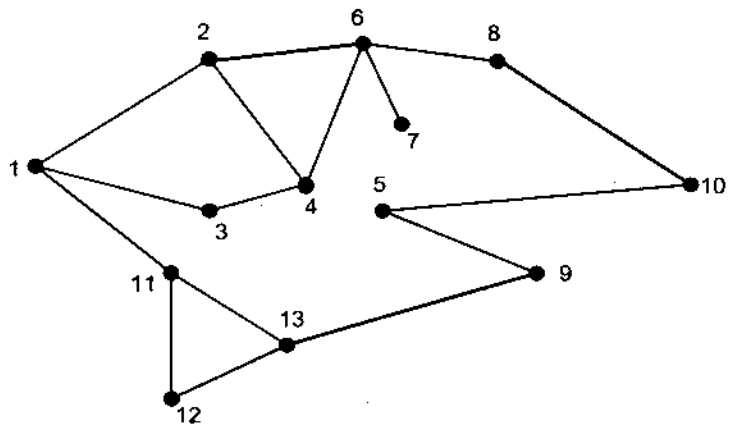
Chương trình cài đặt theo BFS được thể hiện như sau:

```
#include <stdio.h>
#include <conio.h>
#include <io.h>
#include <stdlib.h>
#include <dos.h>
#define MAX 100
#define TRUE 1
#dine FALSE 0
```

```
/* Tìm kiếm theo chiều rộng */
void Init(int A[][MAX], int *n, int *solt, int *chuaxet) {
    FILE *fp; int i, j;
    fp = fopen("BFS.IN", "r");
    if(fp == NULL) {
        printf("\n Không có tệp input");
        delay(2000); return;
    }
    fscanf(fp, "%d", n);
    printf("\n Số đỉnh đồ thị:%d", *n);
    printf("\n Ma trận kề của đồ thị:");
    for(i = 1; i <= *n; i++) {
        printf("\n");
        for(j = 1; j <= *n; j++) {
            fscanf(fp, "%d", &A[i][j]);
            printf("%3d", A[i][j]);
        }
    }
    for(i = 1; i <= *n; i++)
        chuaxet[i] = 0;
    *solt = 0;
}
void BFS(int A[][MAX], int n, int i, int *solt, int chuaxet[], int QUEUE[MAX]) {
    int u, dauQ, cuoiQ, j;
    dauQ = 1; cuoiQ = 1; QUEUE[cuoiQ] = i; chuaxet[i] = *solt;
    while(dauQ <= cuoiQ) {
        u = QUEUE[dauQ]; printf("%3d", u); dauQ = dauQ+1;
        for(j = 1; j <= n; j++) {
            if(A[u][j] == 1 && chuaxet[j] == 0) {
                cuoiQ = cuoiQ+1;
```

```
            QUEUE[cuoiQ] = j;
            chuaxet[j] = *solt;
        }
    }
  }
}
void main(void) {
  int A[MAX][MAX], n, chuaxet[MAX], QUEUE[MAX], solt, i;
  Init(A, &n, &solt, chuaxet);
  printf("\n\n");
  for(i = 1; i <= n; i++)
    if(chuaxet[i] == 0) {
      solt = solt+1;
      BFS(A, n, i, &solt, chuaxet, QUEUE);
    }
  getch();
}
```

Ví dụ. Áp dụng thuật toán tìm kiếm theo chiều rộng với đồ thị trong hình 6.13 sau:



Hình 6.13 Đồ thị vô hướng G

Bước 1. Queue = ϕ;

Bước 2.

- Queue = 1;
- Queue = 1, 2, 3, 11;
- Queue = 1, 2, 3, 11, 6, 4;
- Queue = 1, 2, 3, 11, 4, 6, 12, 13;
- Queue = 1, 2, 3, 11, 4, 6, 12, 13, 7, 8;
- Queue = 1, 2, 3, 11, 4, 6, 12, 13, 7, 8, 9;
- Queue = 1, 2, 3, 11, 4, 6, 12, 13, 7, 8, 9, 10;
- Queue = 1, 2, 3, 11, 4, 6, 12, 13, 7, 8, 9, 10, 5;

6.3.3. Kiểm tra tính liên thông của đồ thị

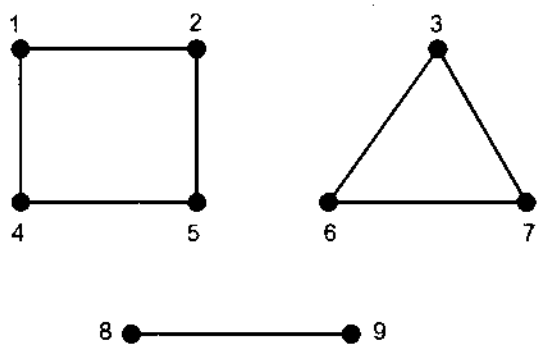
Bài toán: Cho đồ thị G = <V, E>. Trong đó V là tập đỉnh, E là tập cạnh của đồ thị. Hãy tìm số thành phần liên thông của đồ thị và cho biết mỗi thành phần liên thông của đồ thị gồm những đỉnh nào?

Một đồ thị có thể liên thông hoặc có thể không liên thông. Nếu đồ thị là liên thông (số thành phần liên thông là 1), chúng ta chỉ cần gọi tới thủ tục DFS() hoặc BFS() một lần. Nếu đồ thị là không liên thông, khi đó số thành phần liên thông của đồ thị chính bằng số lần gọi tới thủ tục BFS() hoặc DFS(). Để xác định số các thành phần liên thông của đồ thị, chúng ta sử dụng biến mới solt để ghi nhận các đỉnh cùng một thành phần liên thông trong mảng chuaxet:

- Nếu đỉnh i chưa được duyệt, chuaxet[i] có giá trị 0;
- Nếu đỉnh i được duyệt thuộc thành phần liên thông thứ j = solt, ta ghi nhận chuaxet[i] = solt;
- Các đỉnh cùng thành phần liên thông nếu chúng có cùng giá trị trong mảng chuaxet.

```
Procedure BFS(A, n, u, queue, chuaxet);
begin
    queue := ϕ;
    u <= queue;                          (* nạp u vào hàng đợi *)
    solt := solt+1; chuaxet[u] := solt; (* solt là biến toàn cục thiết lập giá trị 0 *)
    while(queue ≠ ϕ) do
    begin
        queue <= p;                      (* lấy p ra từ stack *)
        Tham_Dinh(p);
        for v ∈ ke(p) do
            if(chuaxet[v]) then
            begin
                v <= queue;              (*nạp v vào hàng đợi *)
                chuaxet[v] := solt;
            end;
    end;
end;
```

Ví dụ. Đồ thị vô hướng trong hình 6.14 sẽ cho ta kết quả trong mảng chuaxet như sau:



Hình 6.14. Đồ thị vô hướng G.

BFS(1): 1, 2, 4, 5;

BFS(3): 3, 6, 7;

BFS(8): 8, 9;

chuaxet = {1, 1, 2, 1, 1, 2, 2, 3, 3 }.

Như vậy, đỉnh 1, 2, 4, 5 cùng có giá trị 1 thuộc thành phần liên thông thứ 1;

Đỉnh 3, 6, 7 cùng có giá trị 2 thuộc thành phần liên thông thứ 2;

Đỉnh 8, 9 cùng có giá trị 3 thuộc thành phần liên thông thứ 3.

Chương trình được thể hiện như sau:

```
#include <stdio.h>
#include <conio.h>
#include <io.h>
#include <stdlib.h>
#include <dos.h>
#define MAX    100
#define TRUE 1
#define FALSE 0
/* Tìm kiếm theo chiều rộng */
void Init(int A[][MAX], int *n, int *solt, int *chuaxet) {
   FILE *fp; int i, j;
   fp = fopen("lienth.IN", "r");
   if(fp == NULL) {
      printf("\n Không có tệp input");
      delay(2000); return;
   }
   fscanf(fp, "%d", n);
   printf("\n Số đỉnh đồ thị:%d", *n);
   printf("\n Ma trận kề của đồ thị:");
   for(i = 1; i <= *n; i++) {
      printf("\n");
```

```
        for(j = 1; j <= *n; j++) {
            fscanf(fp, "%d", &A[i][j]);
            printf("%3d", A[i][j]);
        }
    }
    for(i = 1; i <= *n; i++)
        chuaxet[i] = 0;
    *solt = 0;
}
void Result(int *chuaxet, int n, int solt) {
    printf("\n\n");
    if(solt == 1) {
        printf("\n Đồ thị là liên thông");
        getch(); return;
    }
    for(int i = 1; i <= solt; i++) {
        printf("\n Thành phần liên thông thứ %d:", i);
        for(int j = 1; j <= n; j++) {
            if(chuaxet[j] == i)
                printf("%3d", j);
        }
    }
}
void BFS(int A[][MAX], int n, int i, int *solt, int chuaxet[], int QUEUE[MAX]) {
    int u, dauQ, cuoiQ, j;
    dauQ = 1; cuoiQ = 1; QUEUE[cuoiQ] = i; chuaxet[i] = *solt;
    while(dauQ <= cuoiQ) {
        u = QUEUE[dauQ]; printf("%3d", u); dauQ = dauQ+1;
```

```
        for(j = 1; j <= n; j++) {
            if(A[u][j] == 1 && chuaxet[j] == 0) {
                cuoiQ = cuoiQ+1;
                QUEUE[cuoiQ] = j;
                chuaxet[j] = *solt;
            }
        }
    }
}
void Lien_Thong(void) {
    int A[MAX][MAX], n, chuaxet[MAX], QUEUE[MAX], solt, i;
    clrscr(); Init(A, &n, &solt, chuaxet);
    printf("\n\n");
    for(i = 1; i <= n; i++)
        if(chuaxet[i] == 0) {
            solt = solt+1;
            BFS(A, n, i, &solt, chuaxet, QUEUE);
        }
    Result(chuaxet, n, solt);
    getch();
}
void main(void) {
    Lien_Thong();
}
```

6.3.4. Tìm đường đi giữa hai đỉnh bất kỳ của đồ thị

Bài toán: Cho đồ thị G = <V, E>. Trong đó V là tập đỉnh, E là tập cạnh của đồ thị. Hãy tìm đường đi từ đỉnh s $\in$V tới đỉnh t $\in$V của G.

Thủ tục BFS(s) hoặc DFS(s) cho phép ta duyệt các đỉnh cùng một thành phần liên thông với đỉnh s. Như vậy, nếu trong số các đỉnh liên thông với s chứa t thì chắc chắn có đường đi từ đỉnh s đến đỉnh t. Nếu trong số các đỉnh liên thông với đỉnh s không chứa đỉnh t thì không tồn tại đường đi từ đỉnh s đến đỉnh t. Do vậy, chúng ta chỉ cần gọi tới thủ tục DFS(s) hoặc BFS(s) và kiểm tra xem đỉnh t có thuộc thành phần liên thông với s hay không.

```
#include <stdio.h>
#include <conio.h>
#include <io.h>
#include <stdlib.h>
#include <dos.h>
#define MAX 100
#define TRUE 1
#define FALSE 0
int n, truoc[MAX], chuaxet[MAX], queue[MAX];
int A[MAX][MAX]; int s, t;
/* Tìm kiếm theo chiều rộng */
void Init(void) {
    FILE *fp; int i, j;
    fp = fopen("lienth.IN", "r");
    if(fp == NULL) {
        printf("\n Không có tệp input");
        delay(2000); return;
    }
    fscanf(fp, "%d", &n);
    printf("\n Số đỉnh của đồ thị:%d", n);
    printf("\n Ma trận kề của đồ thị:");
    for(i = 1; i <= n; i++) {
```

```
        printf("\n");
        for(j = 1; j <= n; j++) {
            fscanf(fp, "%d", &A[i][j]);
            printf("%3d", A[i][j]);
        }
    }
    for(i = 1; i <= n; i++) {
        chuaxet[i] = TRUE;
        truoc[i] = 0;
    }
}
void Result(void) {
    printf("\n\n");
    if(truoc[t] == 0) {
        printf("\n Không có đường đi từ %d đến %d", s, t);
        getch();
        return;
    }
    printf("\n Đường đi từ %d đến %d là:", s, t);
    int j = t; printf("%d <= ", t);
    while(truoc[j] != s) {
        printf("%3d <= ", truoc[j]);
        j = truoc[j];
    }
    printf("%3d", s);
}
void In(void) {
    printf("\n\n");
```

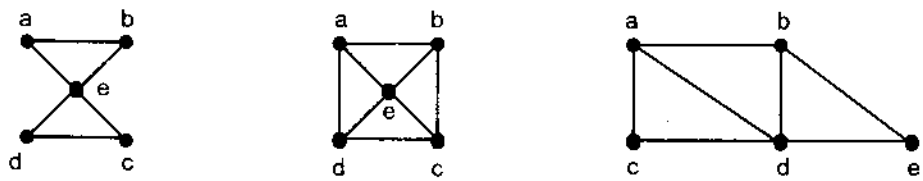
```
    for(int i = 1; i <= n; i++)
        printf("%3d", truoc[i]);
}
void BFS(int s) {
    int dauQ, cuoiQ, p, u; printf("\n");
    dauQ = 1; cuoiQ = 1; queue[dauQ] = s; chuaxet[s] = FALSE;
    while(dauQ <= cuoiQ) {
        u = queue[dauQ]; dauQ = dauQ+1;
        printf("%3d", u);
        for(p = 1; p <= n; p++) {
            if(A[u][p] && chuaxet[p]) {
                cuoiQ = cuoiQ+1; queue[cuoiQ] = p;
                chuaxet[p] = FALSE; truoc[p] = u;
            }
        }
    }
}
void duongdi(void) {
    int chuaxet[MAX], truoc[MAX], queue[MAX];
    Init(); BFS(s); Result();
}
void main(void) {
    clrscr();
    printf("\n Đỉnh đầu:"); scanf("%d", &s);
    printf("\n Đỉnh cuối:"); scanf("%d", &t);
    Init(); printf("\n"); BFS(s);
    n(); getch();
    Result(); getch();
}
```

6.4. ĐƯỜNG ĐI VÀ CHU TRÌNH EULER

Chu trình đơn trong đồ thị G đi qua mỗi cạnh của đồ thị đúng một lần được gọi là chu trình Euler. Đường đi đơn trong G đi qua mỗi cạnh của nó đúng một lần được gọi là đường đi Euler. Đồ thị được gọi là đồ thị Euler nếu nó có chu trình Euler. Đồ thị có đường đi Euler được gọi là nửa Euler.

Rõ ràng, mọi đồ thị Euler đều là nửa Euler nhưng điều ngược lại không đúng.

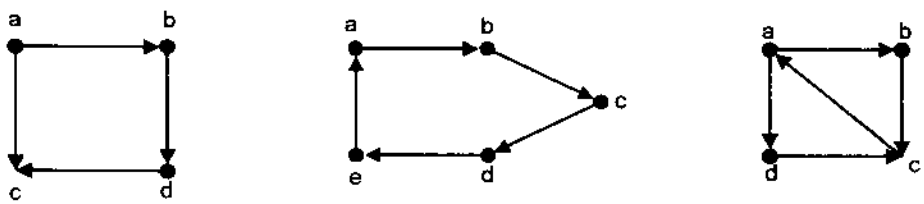
Ví dụ 1. Xét các đồ thị G1, G2, G3 trong hình 6.15.



Hình 6.15. Đồ thị vô hướng G1, G2, G3.

Đồ thị G1 là đồ thị Euler vì nó có chu trình Euler a, e, c, d, e, b, a. Đồ thị G3 không có chu trình Euler nhưng chứa đường đi Euler a, c, d, e, b, d, a, b vì thế G3 là nửa Euler. G2 không có chu trình Euler cũng như đường đi Euler.

Ví dụ 2. Xét các đồ thị có hướng H1, H2, H3 trong hình 6.16.



Hình 6.16. Đồ thị có hướng H1, H2, H3.

Đồ thị H2 là đồ thị Euler vì nó chứa chu trình Euler a, b, c, d, e, a vì vậy nó là đồ thị Euler. Đồ thị H3 không có chu trình Euler nhưng có

đường đi Euler a, b, c, a, d, c nên nó là đồ thị nửa Euler. Đồ thị H1 không chứa chu trình Euler cũng như chu trình Euler.

Định lý. *Đồ thị vô hướng liên thông G = <V, E> là đồ thị Euler khi và chỉ khi mọi đỉnh của G đều có bậc chẵn. Đồ thị vô hướng liên thông G = <V, E> là đồ thị nửa Euler khi và chỉ khi nó không có quá hai đỉnh bậc lẻ.*

Để tìm một chu trình Euler, ta thực hiện theo thuật toán sau:

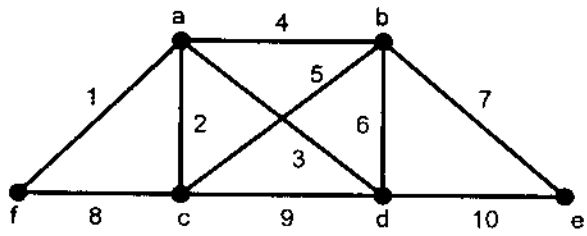
- Tạo một mảng CE để ghi đường đi và một stack để xếp các đỉnh ta sẽ xét. Xếp vào đó một đỉnh tuỳ ý u nào đó của đồ thị, nghĩa là đỉnh u sẽ được xét đầu tiên.
- Xét đỉnh trên cùng của ngăn xếp, giả sử đỉnh đó là đỉnh v và thực hiện:
  - Nếu v là đỉnh cô lập thì lấy v khỏi ngăn xếp và đưa vào CE;
  - Nếu v là liên thông với đỉnh x thì xếp x vào ngăn xếp sau đó xoá bỏ cạnh (v, x);
- Quay lại bước 2 cho tới khi ngăn xếp rỗng thì dừng. Kết quả đường đi Euler được chứa trong CE theo thứ tự ngược lại.

Thủ tục Euler_Cycle sau sẽ cho phép ta tìm chu trình Euler.

```
Procedure   Euler_Cycle;
begin
 Stack := ϕ; CE := ϕ;
 (* Chọn u là đỉnh nào đó của đồ thị *)
 u => Stack;                          { nạp u vào stack };
 while Stack ≠ ϕ do
 begin
    x := top(Stack);                  { x là phần tử đầu stack }
```

```
        if(ke(x) ≠ ϕ) then
        begin
            y := Đỉnh đầu trong danh sách ke(x);
            Stack <= y;                    { nạp y vào Stack };
            Ke(x) := Ke(x) \ (y);
            Ke(y) := Ke(y)\ (x);           {loại cạnh (x, y) khỏi đồ thị };
        end
        else
            x <= Stack;                    { lấy x ra khỏi stack };
            CE <= x;                       { nạp x vào CE; }
    end;
end;
```

Ví dụ. Tìm chu trình Euler trong hình 6.17.



Hình 6.17. Đồ thị vô hướng G.

Các bước thực hiện theo thuật toán sẽ cho ta kết quả như bảng 6.2.

Bảng 6.2 Các bước thực hiện chu trình Euler

Bước	Giá trị trong stack	Giá trị trong CE	Cạnh còn lại
1	F		1, 2, 3, 4, 5, 6, 7, 8, 9, 10
2	f, a		2, 3, 4, 5, 6, 7, 8, 9, 10
3	f, a, c		3, 4, 5, 6, 7, 8, 9, 10
4	f, a, c, f		3, 4, 5, 6, 7, 9, 10

5	f, a, c	f	3, 4, 5, 6, 7, 9, 10
6	f, a, c, b	f	3, 4, 6, 7, 9, 10
7	f, a, c, b, d	f	3, 4, 7, 9, 10
8	f, a, c, b, d, c	f	3, 4, 7, 10
9	f, a, c, b, d	f, c	3, 4, 7, 10
10	f, a, c, b, d, e	f, c	3, 4, 7
11	f, a, c, b, d, e, b	f, c	3, 4
12	f, a, c, b, d, e, b, a	f, c	3
13	f, a, c, b, d, e, b, a, d	f, c	
14	f, a, c, b, d, e, b, a	f, c, d	
15	f, a, c, b, d, e, b	f, c, d, a	
16	f, a, c, b, d, e	f, c, d, a, b	
17	f, a, c, b, d	f, c, d, a, b, e	
18	f, a, c, b	f, c, d, a, b, e, d	
19	f, a, c	f, c, d, a, b, e, d, b	
20	f, a	f, c, d, a, b, e, d, b, c	
21	f	f, c, d, a, b, e, d, b, c, a	
22		f, c, d, a, b, e, d, b, c, a, f	

Chương trình tìm chu trình Euler được thể hiện như sau:

```
#include <stdio.h>

#include <conio.h>

#include <stdlib.h>

#include <math.h>

#include <dos.h>

#define MAX 50

#define TRUE   1

#define FALSE 0

int A[MAX][MAX], n, u = 1;

void Init(void) {

    int i, j; FILE *fp;

    fp = fopen("CTEULER.IN", "r");

    fscanf(fp, "%d", &n);
```

```
    printf("\n Số đỉnh đồ thị:%d", n);
    printf("\n Ma trận kề:");
    for(i = 1; i <= n; i++) {
        printf("\n");
        for(j = 1; j <= n; j++) {
            fscanf(fp, "%d", &A[i][j]);
            printf("%3d", A[i][j]);
        }
    }
    fclose(fp);
}
int Kiemtra(void) {
    int i, j, s, d;
    d = 0;
    for(i = 1; i <= n; i++) {
        s = 0;
        for(j = 1; j <= n; j++)
            s += A[i][j];
        if(s%2) d++;
    }
    if(d>0) return(FALSE);
    return(TRUE);
}
void Tim(void) {
    int v, x, top, dCE;
    int stack[MAX], CE[MAX];
    top = 1; stack[top] = u; dCE = 0;
    do {
```

```
        v = stack[top]; x = 1;
        while(x <= n && A[v][x] == 0)
            x++;
        if(x>n) {
            dCE++; CE[dCE] = v; top--;
        }
        else {
            top++; stack[top] = x;
            A[v][x] = 0; A[x][v] = 0;
        }
    } while(top != 0);
    printf("\n Có chu trình Euler:");

    for(x = dCE; x>0; x--)
        printf("%3d", CE[x]);
}
void main(void) {
    clrscr(); Init();
    if(Kiemtra())
        Tim();
    else printf("\n Không có chu trình Euler");
    getch();
}
```

Một đồ thị không có chu trình Euler nhưng vẫn có thể có đường đi Euler. Khi đó, đồ thị có đúng hai đỉnh bậc lẻ, tức là tổng các số cạnh xuất phát từ một trong hai đỉnh đó là số lẻ. Một đường đi Euler phải xuất phát từ một trong hai đỉnh đó và kết thúc ở đỉnh kia. Như vậy, thuật toán tìm đường đi Euler chỉ khác với thuật toán tìm chu trình

Euler ở chỗ ta phải xác định điểm xuất phát của đường đi. Chương trình tìm đường đi Euler được thể hiện như sau:

```
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
#include <math.h>
#include <dos.h>
#define MAX 50
#define TRUE 1
#define FALSE 0
void Init(int A[][MAX], int *n) {
    int i, j; FILE *fp;
    fp = fopen("DDEULER.IN", "r");
    fscanf(fp, "%d", n);
    printf("\n Số đỉnh đồ thị:%d", *n);
    printf("\n Ma trận kề:");
    for(i = 1; i <= *n; i++) {
        printf("\n");
        for(j = 1; j <= *n; j++) {
            fscanf(fp, "%d", &A[i][j]);
            printf("%3d", A[i][j]);
        }
    }
    fclose(fp);
}
int Kiemtra(int A[][MAX], int n, int *u) {
    int i, j, s, d;
    d = 0;
```

```
    for(i = 1; i <= n; i++) {
        s = 0;
        for(j = 1; j <= n; j++)
            s+ = A[i][j];
        if(s%2) {
            d++; *u = i;
        }
    }
    if(d != 2) return(FALSE);
    return(TRUE);
}
void DDEULER(int A[][MAX], int n, int u) {
    int v, x, top, dCE;
    int stack[MAX], CE[MAX];
    top = 1; stack[top] = u; dCE = 0;
    do {
        v = stack[top]; x = 1;
        while(x <= n && A[v][x] == 0)
            x++;
        if(x>n) {
            dCE++; CE[dCE] = v; top--;
        }
        else {
            top++; stack[top] = x;
            A[v][x] = 0; A[x][v] = 0;
        }
    } while(top != 0);
    printf("\n Có đường đi Euler:");
```

```
    for(x = dCE; x > 0; x--)
        printf("%3d", CE[x]);
}
void main(void) {
    int A[MAX][MAX], n, u;
    clrscr(); Init(A, &n);
    if(Kiemtra(A, n, &u))
        DDEULER(A, n, u);
    else printf("\n Không có đường đi Euler");
    getch();
}
```

Để tìm tất cả các đường đi Euler của một đồ thị n đỉnh, m cạnh, ta có thể dùng kỹ thuật đệ qui như sau:

- Bước 1. Tạo mảng b có độ dài m + 1 như một ngăn xếp chứa đường đi. Đặt b[0] = 1, i = 1 (xét đỉnh thứ nhất của đường đi);
- Bước 2. Lần lượt cho b[i] các giá trị là đỉnh kề với b[i-1] mà cạnh (b[i-1], b[i]) không trùng với những cạnh đã dùng từ b[0] đến b[i-1]. Với mỗi giá trị của b[i], ta kiểm tra:
  - Nếu i < m thì tăng i lên 1 đơn vị (xét đỉnh tiếp theo) và quay lại bước 2.
  - Nếu i = m thì dãy b chính là một đường đi Euler.

Chương trình liệt kê tất cả đường đi Euler được thể hiện như sau:

```
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
#include <math.h>
#include <dos.h>
#define MAX 50
```

```
#define TRUE 1
#define FALSE 0
int m, b[MAX], u, i, OK;
void Init(int A[][MAX], int *n) {
    int i, j, s, d; FILE *fp;
    fp = fopen("DDEULER.IN", "r");
    fscanf(fp, "%d", n);
    printf("\n Số đỉnh đồ thị:%d", *n);
    printf("\n Ma trận kề:");
    u = 1; d = 0; m = 0;
    for(i = 1; i <= *n; i++) {
        printf("\n"); s = 0;
        for(j = 1; j <= *n; j++) {
            fscanf(fp, "%d", &A[i][j]);
            printf("%3d", A[i][j]);
            s+ = A[i][j];
        }
        if(s%2) { d++; u = i; }
        m = m+s;
    }
    m = m /2;
    if(d != 2) OK = FALSE;
    else OK = TRUE;
    fclose(fp);
}
void Result(void) {
    int i;
    printf("\n Có đường đi Euler:");
```

```
    for(i = 0; i <= m; i++)
        printf("%3d", b[i]);
}
void DDEULER(int *b, int A[][MAX], int n, int i) {
    int j, k;
    for(j = 1; j <= n; j++) {
        if(A[b[i-1]][j] == 1) {
            A[b[i-1]][j] = 0; A[j][b[i-1]] = 0;
            b[i] = j;
            if(i == m) Result();
            else DDEULER(b, A, n, i+1);
            A[b[i-1]][j] = 1; A[j][b[i-1]] = 1;
        }
    }
}
void main(void) {
    int A[MAX][MAX], n;
    clrscr(); Init(A, &n);
    b[0] = u; i = 1;
    if(OK) DDEULER(b, A, n, i);
    else printf("\n Không có đường đi Euler");
    getch();
}
```

6.5. ĐƯỜNG ĐI VÀ CHU TRÌNH HAMILTON

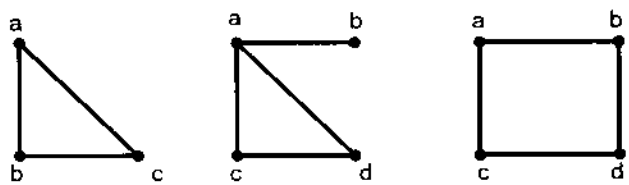
Với đồ thị Euler, chúng ta quan tâm tới việc duyệt các cạnh của đồ thị mỗi cạnh đúng một lần, thì trong mục này, chúng ta xét đến một bài toán tương tự nhưng chỉ khác nhau là ta chỉ quan tâm tới các đỉnh của

đồ thị, mỗi đỉnh đúng một lần. Sự thay đổi này tưởng như không đáng kể, nhưng thực tế có nhiều sự khác biệt trong khi giải quyết bài toán.

Định nghĩa. Đường đi qua tất cả các đỉnh của đồ thị mỗi đỉnh đúng một lần được gọi là đường đi Hamilton. Chu trình bắt đầu tại một đỉnh v nào đó qua tất cả các đỉnh còn lại mỗi đỉnh đúng một lần sau đó quay trở lại v được gọi là chu trình Hamilton. Đồ thị được gọi là đồ thị Hamilton nếu nó chứa chu trình Hamilton. Đồ thị chứa đường đi Hamilton được gọi là đồ thị nửa Hamilton.

Như vậy, một đồ thị Hamilton bao giờ cũng là đồ thị nửa Hamilton nhưng điều ngược lại không luôn luôn đúng. Ví dụ sau sẽ minh họa cho nhận xét này.

Ví dụ. Đồ thị hamilton G3, nửa Hamilton G2 và G1.



Hình 6.18. Đồ thị Hamilton G3, nửa Hamilton G2 và G1.

Cho đến nay, việc tìm ra một tiêu chuẩn để nhận biết đồ thị Hamilton vẫn còn mở, mặc dù đây là vấn đề trung tâm của lý thuyết đồ thị. Hơn thế nữa, cho đến nay cũng vẫn chưa có thuật toán hiệu quả để kiểm tra một đồ thị có phải là đồ thị Hamilton hay không.

Để liệt kê tất cả các chu trình Halmilton của đồ thị, chúng ta có thể sử dụng thuật toán sau:

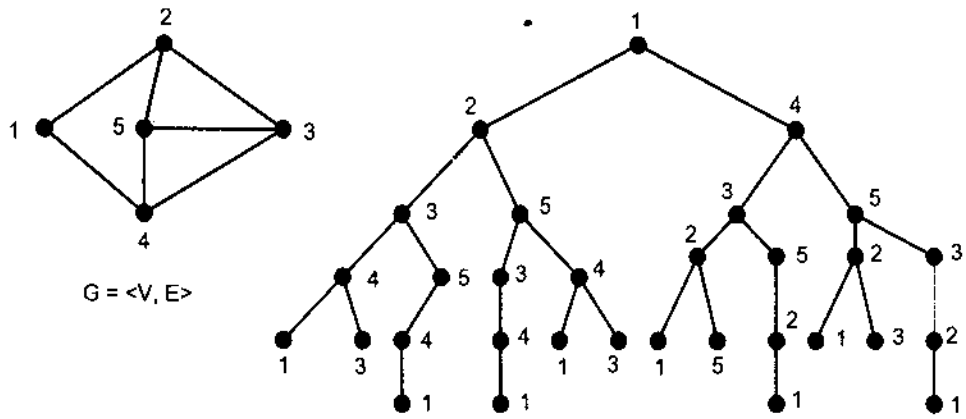
```
Procedure Hamilton(k);
(* Liệt kê các chu trinh hamilton của đồ thị bằng cách phát triển dãy đỉnh
(X[1], X[2],..., X[k-1]) của đồ thị G = (V, E) *)
begin
   for y∈ Ke(X[k-1]) do
```

```
        if(k:= n+1) and (y := v0) then
        Ghinhan(X[1], X[2],..., X[n], v0);
        else
        begin
            X[k]:= y; chuaxet[y] := false;
            Hamilton(k+1);
            chuaxet[y] := true;
        end;
end;
```

Chương trình chính được thể hiện như sau:

```
begin
    for v∈V do chuaxet[v] := true;      (* thiết lập trạng thái các đỉnh *)
    X[1] := v0;                         (* v0 là một đỉnh nào đó của đồ thị *)
    chuaxet[v0] := false;
    Hamilton(2);
end.
```

Cây tìm kiếm chu trình Hamilton thể hiện thuật toán trên được mô tả như trong hình 6.19.



Hình 6.19. Cây tìm kiếm chu trình Hamilton.

Chương trình liệt kê các chu trình Hamilton được thể hiện như sau:

```
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
#include <math.h>
#include <dos.h>
#define MAX 50
#define TRUE 1
#define FALSE 0
int A[MAX][MAX], C[MAX], B[MAX];
int n, i, d;
void Init(void) {
    int i, j; FILE *fp;
    fp = fopen("CCHMTON.IN", "r");
    if(fp == NULL) {
        printf("\n Không có tệp input");
        getch(); return;
    }
    fscanf(fp, "%d", &n);
    printf("\n Số đỉnh đồ thị:%d", n);
    printf("\n Ma trận kề:");
    for(i = 1; i <= n; i++) {
        printf("\n");
        for(j = 1; j <= n; j++) {
            fscanf(fp, "%d", &A[i][j]);
            printf("%3d", A[i][j]);
        }
```

```
    }
    fclose(fp);
    for(i = 1; i <= n; i++)
        C[i] = 0;
}
void Result(void) {
    int i;
    printf("\n ");
    for(i = n; i> = 0; i--)
        printf("%3d", B[i]);
    d++;
}
void Hamilton(int *B, int *C, int i) {
    int j, k;
    for(j = 1; j <= n; j++) {
        if(A[B[i-1]][j] == 1 && C[j] == 0) {
            B[i] = j; C[j] = 1;
            if(i <n) Hamilton(B, C, i+1);
            else if(B[i] == B[0]) Result();
            C[j] = 0;
        }
    }
}
void main(void) {
    B[0] = 1; i = 1; d = 0;
 Init();
    Hamilton(B, C, i);
    if(d == 0)
```

```
        printf("\n Không có chu trình Hamilton");
    getch();
}
```

Chương trình liệt kê tất cả đường đi Hamilton như sau:

```
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
#include <math.h>
#include <dos.h>
#define MAX 50
#define TRUE 1
#define FALSE 0
int A[MAX][MAX], C[MAX], B[MAX];
int n, i, d;
void Init(void) {
    int i, j; FILE *fp;
    fp = fopen("DDHMTON.IN", "r");
    if(fp == NULL) {
        printf("\n Không có tệp input");
        getch(); return;
    }
    fscanf(fp, "%d", &n);
    printf("\n Số đỉnh đồ thị:%d", n);
    printf("\n Ma trận kề:");
    for(i = 1; i <= n; i++) {
        printf("\n");
        for(j = 1; j <= n; j++) {
            fscanf(fp, "%d", &A[i][j]);
```

```
            printf("%3d", A[i][j]);
        }
    }
    fclose(fp);
    for(i = 1; i <= n; i++)
        C[i] = 0;
}
void Result(void) {
    int i;
    printf("\n ");
    for(i = n; i>0; i--)
        printf("%3d", B[i]);
    d++;
}
void Hamilton(int *B, int *C, int i) {
    int j, k;
    for(j = 1; j <= n; j++) {
        if(A[B[i-1]][j] == 1 && C[j] == 0) {
            B[i] = j; C[j] = 1;
            if(i <n) Hamilton(B, C, i+1);
            else Result();
            C[j] = 0;
        }
    }
}
void main(void) {
    B[0] = 1; i = 1; d = 0;
 Init();
```

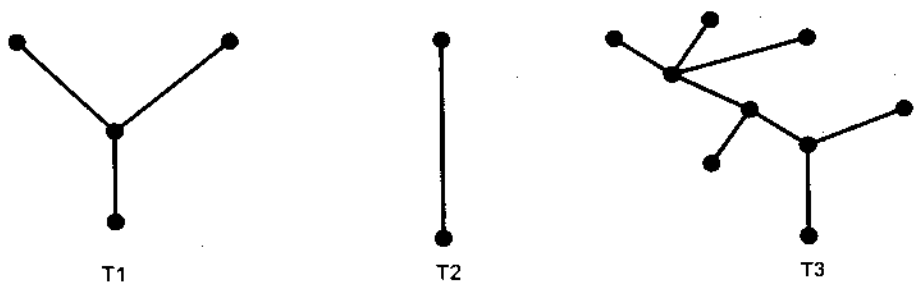
```
    Hamilton(B, C, i);
    if(d == 0)
        printf("\n Không có đường đi Hamilton");
    getch();
}
```

6.6. CÂY BAO TRÙM

Định nghĩa 1. *Ta gọi cây là đồ thị vô hướng liên thông không có chu trình. Đồ thị không có chu trình được gọi là rừng.*

Như vậy, rừng là đồ thị mà mỗi thành phần liên thông của nó là một cây.

Ví dụ. Rừng gồm 3 cây trong hình 6.20:



Hình 6.20 Rừng gồm 3 cây T1, T2, T3.

Cây được coi là dạng đồ thị đơn giản nhất của đồ thị. Định lý sau đây cho ta một số tính chất của cây.

Định lý. *Giả sử G = <V, E> là đồ thị vô hướng n đỉnh. Khi đó những khẳng định sau là tương đương:*

a) G là một cây.

b) G là đồ thị vô hướng liên thông không có chu trình.

c) G liên thông và có đúng n-1 cạnh.

d) Giữa hai đỉnh bất kỳ của G có đúng một đường đi.

e) G liên thông và mỗi cạnh của nó đều là cầu.

f) G không chứa chu trình nhưng hễ cứ thêm vào nó một cạnh ta thu được đúng một chu trình.

Định nghĩa 2. *Cho G là đồ thị vô hướng liên thông. Ta gọi đồ thị con T của G là một cây bao trùm nếu T thoả mãn hai điều kiện:*

a) T là một cây;

b) Tập đỉnh của T bằng tập đỉnh của G.

Đối với cây bao trùm, chúng ta quan tâm tới những bài toán cơ bản sau:

Bài toán 1. Cho G = <V, E> là đồ thị vô hướng liên thông. Hãy xây dựng một cây bao trùm của G.

Bài toán 2. Cho G = <V, E> là đồ thị vô hướng liên thông có trọng số. Hãy tìm cây bao trùm nhỏ nhất của G.

6.1.1. Tìm một cây bao trùm trên đồ thị

Để tìm một cây bao trùm trên đồ thị vô hướng liên thông, có thể sử dụng kỹ thuật tìm kiếm theo chiều rộng hoặc tìm kiếm theo chiều sâu để thực hiện. Giả sử ta cần xây dựng một cây bao trùm xuất phát tại đỉnh u nào đó. Trong cả hai trường hợp, mỗi khi ta đến được đỉnh v tức (chuaxet[v] = TRUE) từ đỉnh u thì cạnh (u, v) được kết nạp vào cây bao trùm. Hai kỹ thuật này được thể hiện trong hai thủ tục STREE_DFS(u) và STREE_BFS(v) như sau:

```
Procedure STREE_DFS(u);
begin
/* Tìm kiếm theo chiều sâu, áp dụng cho bài toán xây dựng cây bao trùm
của đồ thị vô hướng liên thông G = (V, E); các biến chuaxet, Ke, T là toàn
cục */
chuaxet[u] = TRUE;
for v ∈ Ke(u) do
```

```
begin
        if chuaxet[v] then
        begin
            T := T ∪ (u, v);
            STREE_DFS(v);
        end;
end;
end;
/* Chương trình chính */
begin
    for u∈V do
        chuaxet[u] := TRUE;
    T := ϕ;
    STREE_DFS(root);            /* root là một đỉnh nào đó của đồ thị */
end.
Procedure STREE_BFS(u);
begin
    QUUE := ϕ;
QUEUE <= u;                     /* đưa u vào hàng đợi*/
chuaxet[u] := false;
while QUEUE≠ ϕ do
    begin
        v <= QUEUE;             /* lấy v khỏi hàng đợi */
        for p ∈ Ke(v) do
        begin
            if chuaxet[u] then
            begin
                QUEUE <= u; chuaxet[u] := FALSE;
                T = T∪(v, p);
```

```
            end;
        end;
    end;
end;
/* Chương trình chính */
begin
    for u ∈ V do
        chuaxet[u] := TRUE;
    T := ϕ;
    STREE-BFS(root);
end.
```

Chương trình xây dựng một cây bao trùm được thể hiện như sau:

```
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
#include <math.h>
#include <dos.h>
#define MAX 50
#define TRUE   1
#define FALSE 0
int CBT[MAX][2], n, A[MAX][MAX], chuaxet[MAX], sc, QUEUE[MAX];
void Init(void) {
    int i, j; FILE *fp;
    fp = fopen("BAOTRUM1.IN", "r");
    if(fp == NULL) {
        printf("\n Không có tệp input");
        getch(); return;
    }
    fscanf(fp, "%d", &n);
```

```
    printf("\n Số đỉnh đồ thị:%d", n);
    printf("\n Ma trận kề");
    for(i = 1; i <= n; i++) {
        printf("\n");
        for(j = 1; j <= n; j++) {
            fscanf(fp, "%d", &A[i][j]);
            printf("%3d", A[i][j]);
        }
    }
    fclose(fp);
    for(i = 1; i <= n; i++)
        chuaxet[i] = TRUE;
}
void STREE-DFS(int i) {
    int j;
    if(sc == n-1) return;
    for(j = 1; j <= n; j++) {
        if(chuaxet[j] && A[i][j]) {
            chuaxet[j] = FALSE; sc++;
            CBT[sc][1] = i; CBT[sc][2] = j;
            if(sc == n-1) return;
            STREE-DFS(j);
        }
    }
}
void Result(void) {
    int i, j;
    for(i = 1; i <= sc; i++) {
        printf("\n Cạnh %d:", i);
```

```
        for(j = 1; j <= 2; j++)
            printf("%3d", CBT[i][j]);
    }
    getch();
}
void STREE_BFS(int u) {
    int dauQ, cuoiQ, v, p;
    dauQ = 1; cuoiQ = 1; QUEUE[dauQ] = u; chuaxet[u] = FALSE;
    while(dauQ <= cuoiQ) {
        v = QUEUE[dauQ]; dauQ = dauQ+1;
        for(p = 1; p <= n; p++) {
            if(chuaxet[p] && A[v][p]) {
                chuaxet[p] = FALSE; sc++;
                CBT[sc][1] = v; CBT[sc][2] = p;
                cuoiQ = cuoiQ+1;
                QUEUE[cuoiQ] = p;
                if(sc == n-1) return;
            }
        }
    }
}
void main(void) {
    int i; Init(); sc = 0; i = 1; chuaxet[i] = FALSE;
                                        /* xây dựng cây bao trùm tại đỉnh 1 */
    STREE_BFS(i);                       /* STREE_DFS(i) */
Result(); getch();
}
```

6.6.2. Tìm cây bao trùm ngắn nhất

Bài toán tìm cây bao trùm nhỏ nhất là một trong những bài toán tối ưu trên đồ thị có ứng dụng trong nhiều lĩnh vực khác nhau của thực tế. Bài toán được phát biểu như sau:

Cho G = (V, E) là đồ thị vô hướng liên thông với tập đỉnh V = {1, 2,..., n} và tập cạnh E gồm m cạnh. Mỗi cạnh e của đồ thị được gán với một số không âm c(e) được gọi là độ dài của nó. Giả sử H = (V, T) là một cây bao trùm của đồ thị G. Ta gọi độ dài c(H) của cây bao trùm H là tổng độ dài các cạnh của nó: $c(H) = \sum_{e \in T} c(e)$. Bài toán được đặt ra là, trong số các cây khung của đồ thị hãy tìm cây khung có độ dài nhỏ nhất của đồ thị.

Để minh họa cho những ứng dụng của bài toán này, chúng ta có thể tham khảo hai mô hình thực tế của bài toán.

Bài toán nối mạng máy tính. Một mạng máy tính gồm n máy tính được đánh số từ 1, 2,..., n. Biết chi phí nối máy i với máy j là c[i, j], i, j = 1, 2,..., n. Hãy tìm cách nối mạng sao cho chi phí là nhỏ nhất.

Bài toán xây dựng hệ thống cable. Giả sử ta muốn xây dựng một hệ thống cable điện thoại nối n điểm của một mạng viễn thông sao cho điểm bất kỳ nào trong mạng đều có đường truyền tin tới các điểm khác. Biết chi phí xây dựng hệ thống cable từ điểm i đến điểm j là c[i, j]. Hãy tìm cách xây dựng hệ thống mạng cable sao cho chi phí là nhỏ nhất.

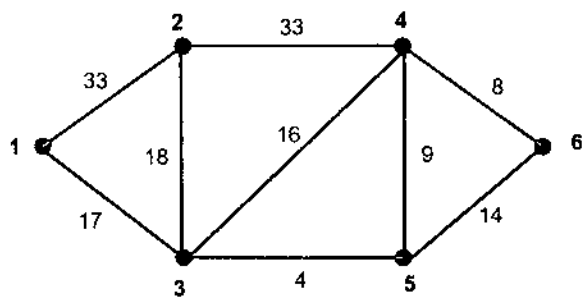
Để giải bài toán cây bao trùm nhỏ nhất, chúng ta có thể liệt kê toàn bộ cây bao trùm và chọn trong số đó một cây nhỏ nhất. Phương án như vậy thực sự không khả thi vì số cây bao trùm của đồ thị là rất lớn cỡ n^{n-2}, điều này không thể thực hiện được với đồ thị với số đỉnh cỡ chục.

Để tìm một cây bao trùm chúng ta có thể thực hiện theo các bước như sau:

- Bước 1. Thiết lập tập cạnh của cây bao trùm là ϕ. Chọn cạnh e = (i, j) có độ dài nhỏ nhất bổ sung vào T.

- Bước 2. Trong số các cạnh thuộc $E \setminus T$, tìm cạnh $e = (i_1, j_1)$ có độ dài nhỏ nhất sao cho khi bổ sung cạnh đó vào T không tạo nên chu trình. Để thực hiện điều này, chúng ta phải chọn cạnh có độ dài nhỏ nhất sao cho hoặc $i_1 \in T$ và $j_1 \notin T$, hoặc $j_1 \in T$ và $i_1 \notin T$.
- Bước 3. Kiểm tra xem T đã đủ n-1 cạnh hay chưa? Nếu T đủ n-1 cạnh thì nó chính là cây bao trùm ngắn nhất cần tìm. Nếu chưa đủ n-1 cạnh thì thực hiện lại bước 2.

Ví dụ. Tìm cây khung nhỏ nhất của đồ thị trong hình 6.21.



Hình 6.21. Đồ thị vô hướng liên thông G = <V, E>

- Bước 1. Đặt T = ϕ. Chọn cạnh (3, 5) có độ dài nhỏ nhất bổ sung vào T.
- Bước 2. Sau ba lần lặp đầu tiên, ta lần lượt bổ sung vào các cạnh (4, 5), (4, 6). Rõ ràng, nếu bổ sung vào cạnh (5, 6) sẽ tạo nên chu trình vì đỉnh 5, 6 đã có mặt trong T. Tình huống tương tự cũng xảy ra đối với cạnh (3, 4) là cạnh tiếp theo của dãy. Tiếp đó, ta bổ sung hai cạnh (1, 3), (2, 3) vào T.
- Bước 3. Tập cạnh trong T đã đủ n-1 cạnh: T = {(3, 5), (4, 6), (4, 5), (1, 3), (2, 3)} chính là cây bao trùm ngắn nhất.

Chương trình tìm cây bao trùm ngắn nhất được thể hiện như sau:

```
#include <stdio.h>

#include <conio.h>

#include <stdlib.h>
```

```
#include <math.h>
#include <dos.h>
#define MAX 50
#define TRUE 1
#define FALSE 0
int E1[MAX], E2[MAX], D[MAX], EB[MAX], V[MAX];
/*  E1: Lưu trữ tập đỉnh đầu của các cạnh;
    E2: Lưu trữ tập đỉnh cuối của các cạnh;
    D: Độ dài các cạnh;
    EB: Tập cạnh cây bao trùm;
    V: Tập đỉnh của đồ thị cũng là tập đỉnh của cây bao trùm;
*/
int i, k, n, m, sc, min, dai;
FILE *fp;
void Init(void) {
    fp = fopen("BAOTRUM.IN", "r");
    if(fp == NULL) {
        printf("\n Không có tệp input");
        getch(); return;
    }
    fscanf(fp, "%d%d", &n, &m);
    printf("\n Số đỉnh đồ thị:%d", n);
    printf("\n Số cạnh đồ thị:%d", m);
    printf("\n Danh sách cạnh:");
    for(i = 1; i <= m; i++) {
        fscanf(fp, "%d%d%d", &E1[i], &E2[i], &D[i]);
        printf("\n%4d%4d%4d", E1[i], E2[i], D[i]);
    }
    fclose(fp);
```

```
    for(i = 1; i <= m; i++) EB[i] = FALSE;
    for(i = 1; i <= n; i++) V[i] = FALSE;
}
void STREE_SHORTEST(void) {
    /* Giai đoạn 1 của thuật toán là tìm cạnh k có độ dài nhỏ nhất */
    min = D[1]; k = 1;
    for(i = 2; i <= m; i++) {
        if(D[i] <min) {
            min = D[i]; k = i;
        }
    }
    /* Kết nạp cạnh k vào cây bao trùm */
     EB[k] = TRUE; V[E1[k]] = TRUE; V[E2[k]] = TRUE; sc = 1;
    do {
        min = 32000;
        for(i = 1; i <= m; i++) {
            if(EB[i] == FALSE && (
                ((V[E1[i]]) && (V[E2[i]] == FALSE))||
                ((V[E1[i]] == FALSE) && (V[E2[i]] == TRUE)))
                && (D[i] <min)) {
                    min = D[i]; k = i;
            }
        }
      /* Tìm k là cạnh nhỏ nhất thỏa mãn điều kiện nếu kết nạp cạnh vào cây sẽ
      không tạo nên chu trình*/
        EB[k] = TRUE; V[E1[k]] = TRUE; V[E2[k]] = TRUE; sc = sc+1;
    }while(sc != (n-1));
}
```

```
void Result(void) {
    printf("\n Cây bao trùm:");
    dai = 0;
    for(i = 1; i <= m; i++) {
        if(EB[i]) {
            printf("\n Cạnh %4d %4d dài %4d", E1[i], E2[i], D[i]);
            dai = dai+D[i];
        }
    }
    printf("\n Đồ dài cây bao trùm:%d", dai);
}
void main(void) {
    Init();
    STREE_SHORTEST();
    Result();
    getch();
}
```

Chúng ta sẽ xét một vài thuật toán hiệu quả hơn. Đó là thuật toán Kruskal và thuật toán Prim.

6.6.3. Thuật toán Kruskal

Thuật toán sẽ xây dựng tập cạnh T của cây khung nhỏ nhất H = <V, T> theo từng bước như sau:

a) Sắp xếp các cạnh của đồ thị G theo thứ tự tăng dần của trọng số cạnh;

b) Xuất phát từ tập cạnh T = ϕ, ở mỗi bước, ta sẽ lần lượt duyệt trong danh sách các cạnh đã được sắp xếp, từ cạnh có trọng số nhỏ đến cạnh có trọng số lớn để tìm ra cạnh mà khi bổ

sung nó vào T không tạo thành chu trình trong tập các cạnh đã được bổ sung vào T trước đó;

c) Thuật toán sẽ kết thúc khi ta thu được tập T gồm n-1 cạnh.

Thuật toán được mô tả thông qua thủ tục Kruskal như sau:

```
Procedure Kruskal;
begin
    T = ϕ;
    while | T | < (n-1) and (E≠ ϕ) do
    begin
        Chọn cạnh e ∈E là cạnh có độ dài nhỏ nhất;
        E := E\ {e};
        if(T ∪ {e}: không tạo nên chu trình ) then T := T ∪ {e};
    end;
    if(| T | <n-1) then (Đồ thị không liên thông);
end;
```

Chương trình tìm cây khung nhỏ nhất theo thuật toán Kruskal được thể hiện như sau:

```
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
#include <math.h>
#include <dos.h>
#define MAX 50
#define TRUE 1
#define FALSE 0
int n, m, minl, connect;
int dau[500], cuoi[500], w[500];
```

```
int daut[50], cuoit[50], father[50];
void Init(void) {
    int i; FILE *fp;
    fp = fopen("baotrum1.in", "r");
    fscanf(fp, "%d%d", &n, &m);
    printf("\n Số đỉnh đồ thị:%d", n);
    printf("\n Số cạnh đồ thị:%d", m);
    printf("\n Danh sách kề đồ thị:");
    for(i = 1; i <= m; i++) {
        fscanf(fp, "%d%d%d", &dau[i], &cuoi[i], &w[i]);
        printf("\n Cạnh %d: %5d%5d%5d", i, dau[i], cuoi[i], w[i]);
    }
    fclose(fp); getch();
}
void Heap(int First, int Last) {
    int j, k, t1, t2, t3;
    j = First;
    while(j <= (Last/2)) {
        if((2*j) <Last && w[2*j + 1] < w[2*j])
            k = 2*j +1;
        else
            k = 2*j;
        if(w[k] < w[j]) {
            t1 = dau[j]; t2 = cuoi[j]; t3 = w[j];
            dau[j] = dau[k]; cuoi[j] = cuoi[k]; w[j] = w[k];
            dau[k] = t1; cuoi[k] = t2; w[k] = t3;
            j = k;
        }
```

```
        else j = Last;
    }
}
int Find(int i) {
    int tro = i;
    while(father[tro]>0)
        tro = father[tro];
    return(tro);
}
void Union(int i, int j) {
    int x = father[i]+father[j];
    if(father[i]>father[j]) {
        father[i] = j;
        father[j] = x;
    }
    else {
        father[j] = i;
        father[i] = x;
    }
}
void Krusal(void) {
    int i, last, u, v, r1, r2, ncanh, ndinh;
    for(i = 1; i <= n; i++)
        father[i] = -1;
    for(i = m/2; i>0; i++)
        Heap(i, m);
    last = m; ncanh = 0; ndinh = 0; minl = 0; connect = TRUE;
    while(ndinh <n-1 && ncanh <m) {
```

```
        ncanh = ncanh+1;
        u = dau[1]; v = cuoi[1];
        r1 = Find(u); r2 = Find(v);
        if(r1 != r2) {
            ndinh = ndinh+1; Union(r1, r2);
            daut[ndinh] = u; cuoit[ndinh] = v;
            minl = minl+w[1];
        }
        dau[1] = dau[last];
        cuoi[1] = cuoi[last];
        w[1] = w[last];
        last = last-1;
        Heap(1, last);
    }
    if(ndinh != n-1) connect = FALSE;
}
void Result(void) {
    int i;
    printf("\n Độ dài cây khung nhỏ nhất:%d", minl);
    printf("\n Các cạnh của cây khung nhỏ nhất:");
    for(i = 1; i <n; i++)
        printf("\n %5d%5d", daut[i], cuoit[i]);
    printf("\n");
}
void main(void) {
    clrscr(); Init();
    Krusal(); Result(); getch();
}
```

6.6.4. Thuật toán Prim

Thuật toán Kruskal làm việc kém hiệu quả đối với những đồ thị có số cạnh khoảng m = n(n-1)/2. Trong những tình huống như vậy, thuật toán Prim tỏ ra hiệu quả hơn. Thuật toán Prim còn được mang tên là người láng giềng gần nhất. Trong thuật toán này, bắt đầu tại một đỉnh tuỳ ý s của đồ thị, nối s với đỉnh y sao cho trọng số cạnh c[s, y] là nhỏ nhất. Tiếp theo, từ đỉnh s hoặc y tìm cạnh có độ dài nhỏ nhất, điều này dẫn đến đỉnh thứ ba z và ta thu được cây bộ phận gồm 3 đỉnh 2 cạnh. Quá trình được tiếp tục cho tới khi ta nhận được cây gồm n-1 cạnh, đó chính là cây bao trùm nhỏ nhất cần tìm.

Trong quá trình thực hiện thuật toán, ở mỗi bước, ta có thể nhanh chóng chọn đỉnh và cạnh cần bổ sung vào cây khung, các đỉnh của đồ thị được sẽ được gán các nhãn. Nhãn của một đỉnh v gồm hai phần, [d[v], near[v]]. Trong đó, phần thứ nhất d[v] dùng để ghi nhận độ dài cạnh nhỏ nhất trong số các cạnh nối đỉnh v với các đỉnh của cây khung đang xây dựng. Phần thứ hai, near[v] ghi nhận đỉnh của cây khung gần v nhất. Thuật toán Prim được mô tả thông qua thủ tục sau:

```
Procedure Prim;
begin
   (* Bước khởi tạo *)
Chọn s là một đỉnh nào đó của đồ thị;
V_H := {s}; T := ϕ; d[s] := 0; near[s] := s;
for v∈ V\V_H do
begin
   D[v] := C[s, v]; near[v] := s;
end;
(* Bước lặp *)
Stop := False;
while not stop do
begin
```

```
        Tìm u∈ V\V_H thoả mãn: d[u] = min {d[v] với u∈V\V_H};
        V_H = V_H∪ {u}; T = T ∪ {u, near[u]};
        if | V_H | = n then
        begin
            H = (V_H, T) là cây khung nhỏ nhất của đồ thị;
            Stop := TRUE;
        end
        else
            for v ∈ V\V_H do
                if d[v] > C[u, v] then
                begin
                    D[v] := C[u, v];
                    Near[v] := u;
                end;
    end;
    end;
```

Chương trình cài đặt thuật toán Prim tìm cây bao trùm nhỏ nhất được thực hiện như sau:

```
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
#include <math.h>
#include <dos.h>
#define TRUE 1
#define FALSE 0
#define MAX 10000
int a[100][100];
int n, m, i, sc, w;
```

```
int chuaxet[100];
int cbt[100][3];
FILE *f;
void nhap(void) {
    int p, i, j, k;
    for(i = 1; i <= n; i++)
        for(j = 1; j <= n; j++)
            a[i][j] = 0;
    f = fopen("baotrum.in", "r");
    fscanf(f, "%d%d", &n, &m);
    printf("\n Số đỉnh: %3d ", n);
    printf("\n Số cạnh: %3d", m);
    printf("\n Danh sách cạnh:");
    for(p = 1; p <= m; p++) {
        fscanf(f, "%d%d%d", &i, &j, &k);
        printf("\n %3d%3d%3d", i, j, k);
        a[i][j] = k; a[j][i] = k;
    }
    for(i = 1; i <= n; i++) {
        printf("\n");
        for(j = 1; j <= n; j++) {
            if(i != j && a[i][j] == 0)
                a[i][j] = MAX;
            printf("%7d", a[i][j]);
        }
    }
    fclose(f); getch();
}
```

```
void Result(void) {
    for(i = 1; i <= sc; i++)
        printf("\n %3d%3d", cbt[i][1], cbt[i][2]);
}
void PRIM(void) {
    int i, j, k, top, min, l, t, u;
    int s[100];
    sc = 0; w = 0; u = 1;
    for(i = 1; i <= n; i++)
        chuaxet[i] = TRUE;
    top = 1; s[top] = u;
    chuaxet[u] = FALSE;
    while(sc <n-1) {
        min = MAX;
        for(i = 1; i <= top; i++) {
            t = s[i];
            for(j = 1; j <= n; j++) {
                if(chuaxet[j] && min>a[t][j]) {
                    min = a[t][j];
                    k = t; l = j;
                }
            }
        }
        sc++; w = w+min;
        cbt[sc][1] = k; cbt[sc][2] = l;
        chuaxet[l] = FALSE; a[k][l] = MAX;
        a[l][k] = MAX; top++; s[top] = l;
        printf("\n");
```

```
    }
}
void main(void) {
    clrscr();
    nhap(); PRIM();
    printf("\n Độ dài ngắn nhất:%d", w);
    for(i = 1; i <= sc; i++)
      printf("\n %3d%3d", cbt[i][1], cbt[i][2]);
    getch();
}
```

6.7. BÀI TOÁN TÌM ĐƯỜNG ĐI NGẮN NHẤT

Xét đồ thị có hướng G = <V, E>; trong đó | V| = n, | E | = m. Với mỗi cung(u, v) ∈ E, ta đặt tương ứng với nó một số thực A(u, v) được gọi là trọng số của cung. Ta sẽ đặt A[u, v] = ∞ nếu (u, v) ∉ E. Nếu dãy $v_0, v_1, \ldots, v_k$ là một đường đi trên G thì $\sum_{i=1}^{p} A[v_{i-1}, v_i]$ được gọi là độ dài của đường đi.

Bài toán tìm đường đi ngắn nhất trên đồ thị có hướng dưới dạng tổng quát có thể được phát biểu dưới dạng sau: tìm đường đi ngắn nhất từ một đỉnh xuất phát s∈V (đỉnh nguồn) đến đỉnh cuối t∈V (đỉnh đích). Đường đi như vậy được gọi là đường đi ngắn nhất từ s đến t, độ dài của đường đi d(s, t) được gọi là khoảng cách ngắn nhất từ s đến t (trong trường hợp tổng quát d(s, t) có thể âm). Nếu như không tồn tại đường đi từ s đến t thì độ dài đường đi d(s, t) = ∞. Nếu như mỗi chu trình trong đồ thị đều có độ dài dương thì trong đường đi ngắn nhất sẽ không có đỉnh nào bị lặp lại, đường đi như vậy được gọi là đường đi cơ bản. Nếu như đồ thị tồn tại một chu trình nào đó có độ dài âm, thì đường đi ngắn nhất có thể không xác định, vì ta có thể đi qua chu trình

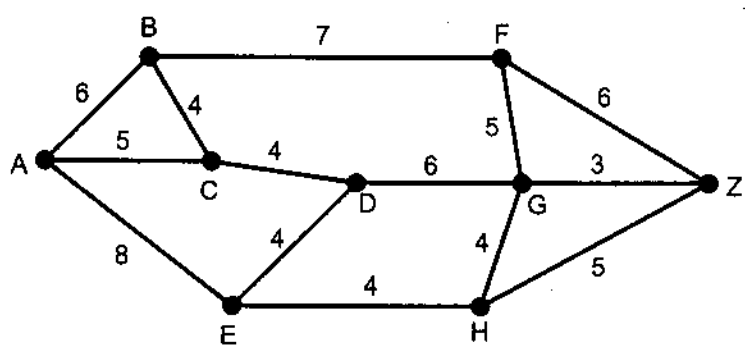
âm đó một số lần đủ lớn để độ dài của nó nhỏ hơn bất kỳ một số thực cho trước nào.

6.1.1. Thuật toán gán nhãn

Có rất nhiều thuật toán khác nhau được xây dựng để tìm đường đi ngắn nhất. Nhưng tư tưởng chung của các thuật toán đó có thể được mô tả như sau:

Từ ma trận trọng số A[u, v], u, v∈V, ta tìm cận trên d[v] của khoảng cách từ s đến tất cả các đỉnh v∈V. Mỗi khi phát hiện thấy d[u] + A[u, v] < d[v] thì cận trên d[v] sẽ được làm tốt lên bằng cách d[v] = d[u] + A[u, v]. Quá trình sẽ kết thúc khi nào ta không thể làm tốt hơn lên được bất kỳ cận trên nào, khi đó d[v] sẽ cho ta giá trị ngắn nhất từ đỉnh s đến đỉnh v. Giá trị d[v] được gọi là nhãn của đỉnh v. Tư tưởng trên có được thể hiện bằng một thuật toán gán nhãn tổng quát như sau:

Ví dụ 1. Tìm đường đi ngắn nhất từ đỉnh A đến đỉnh Z trên đồ thị hình 6.22.



Hình 6.22. Đồ thị trọng số G

- Bước 1. Gán cho nhãn đỉnh A là 0;
- Bước 2. Trong số các cạnh (cung) xuất phát từ A, ta chọn cạnh có độ dài nhỏ nhất, sau đó gán nhãn cho đỉnh đó bằng nhãn của đỉnh A cộng với độ

dài cạnh tương ứng. Ta chọn được đỉnh C có trọng số AC = 5, nhãn d[C] = 0 + 5 = 5.

- Bước 3. Tiếp đó, trong số các cạnh (cung) đi từ một đỉnh có nhãn là A hoặc C tới một đỉnh chưa được gán nhãn, ta chọn cạnh (cung) sao cho nhãn của đỉnh cộng với trọng số cạnh tương ứng là nhỏ nhất gán cho nhãn của đỉnh cuối của cạnh (cung). Như vậy, ta lần lượt gán được các nhãn như sau: d[B] = 6 vì d[B] < d[C] + | CB| = 5 + 4; d[E] = 8; Tiếp tục làm như vậy cho tới khi đỉnh Z được gán nhãn đó chính là độ dài đường đi ngắn nhất từ A đến Z. Thực chất, nhãn của mỗi đỉnh chính là đường đi ngắn nhất từ đỉnh nguồn tới nó. Quá trình có thể được mô tả như trong bảng dưới đây.

Bảng 6.3 Quá trình tìm đường đi ngắn nhất.

Bước	Đỉnh được gán nhãn	Nhãn các đỉnh	Đỉnh đã dùng để gán nhãn
Khởi tạo	A	0	
1	C	0 + 5 = 5	A
2	B	0 + 6 = 6	A
3	E	0 + 8 = 8	A
4	D	5 + 4 = 9	C
5	F	6 + 7 = 13	B
6	H	8 + 6 = 14	E
7	G	9 + 6 = 15	D
8	Z	15 + 3 = 18	Z

Như vậy, độ dài đường đi ngắn nhất từ A đến Z là 18. Đường đi ngắn nhất từ A đến Z qua các đỉnh: A -> C-> D -> G -> Z.

Chú ý rằng, thuật toán gán nhãn có thể áp dụng cho cả đồ thị vô hướng hoặc có hướng.

6.7. 2. Thuật toán Dijkstra

Thuật toán tìm đường đi ngắn nhất từ đỉnh s đến các đỉnh còn lại được Dijkstra đề nghị áp dụng cho trường hợp đồ thị có trọng số không âm. Thuật toán được thực hiện trên cơ sở gán tạm thời cho các đỉnh.

Nhãn của mỗi đỉnh cho biết cận trên của độ dài đường đi ngắn nhất tới đỉnh đó. Các nhãn này sẽ được biến đổi (tính lại) nhờ một thủ tục lặp, mà ở mỗi bước lặp một số đỉnh sẽ có nhãn không thay đổi, nhãn đó chính là độ dài đường đi ngắn nhất từ s đến đỉnh đó. Thuật toán có thể được mô tả bằng thủ tục Dijkstra như sau:

```
Procedure Dijkstra;
(*Đầu vào G = <V, E> với n đỉnh có ma trận trọng số A[u, v] ≥ 0; s ∈ V *)
(*Đầu ra là khoảng cách nhỏ nhất từ s đến các đỉnh còn lại d[v]: v ∈ V.
Truoc[v] ghi lại đỉnh trước v trong đường đi ngắn nhất từ s đến v*)
begin
    (* Bước 1: Khởi tạo nhãn tạm thời cho các đỉnh*)
for v∈ V do
begin
    d[v] := A[s, v];
    truoc[v]:= s;
end;
d[s]:= 0; T := V\ {s};                    (* T là tập đỉnh có nhãn tạm thời *)
while T != ϕ do                           (* bước lặp *)
begin
    Tìm đỉnh u∈T sao cho d[u] = min { d[z]: z∈T}
    T:= T\ {u};                           (* cố định nhãn đỉnh u *);
    for v∈T do                            (* Gán lại nhãn cho các đỉnh trong T*)
    begin
        if(d[v] > d[u] + A[u, v]) then
        begin
            d[v] := d[u] + A[u, v];
            truoc[v] := u;
        end;
```

```
        end;
    end;
    end.
```

Chương trình cài đặt thuật toán Dijkstra tìm đường đi ngắn nhất từ một đỉnh đến tất cả các đỉnh khác của đồ thị có hướng với trọng số không âm được thực hiện như sau:

```
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
#include <math.h>
#include <dos.h>
#define MAX 50
#define TRUE 1
#define FALSE 0
int n, s, t;
char chon;
int truoc[MAX], d[MAX], CP[MAX][MAX];
int final[MAX];
void Init(void) {
    FILE * fp; int i, j;
    fp = fopen("ijk1.in", "r");
    fscanf(fp, "%d", &n);
    printf("\n Số đỉnh:%d", n);
    printf("\n Ma trận khoảng cách:");
    for(i = 1; i <= n; i++) {
        printf("\n");
        for(j = 1; j <= n; j++) {
            fscanf(fp, "%d", &CP[i][j]);
```

```
            printf("%3d", CP[i][j]);
            if(CP[i][j] == 0) CP[i][j] = 32000;
        }
    }
    fclose(fp);
}
void Result(void) {
    int i, j;
    printf("\n Đường đi ngắn nhất từ %d đến %d là \n", s, t);
    printf("%d <= ", t);
    i = truoc[t];
    while(i != s) {
        printf("%d <= ", i);
        i = truoc[i];
    }
    printf("%d", s);
    printf("\n Độ dài đường đi là: %d", d[t]);
    getch();
}
void Dijkstra(void) {
    int v, u, minp;
    printf("\n Tìm đường đi từ s = "); scanf("%d", &s);
    printf(" đến "); scanf("%d", &t);
    for(v = 1; v <= n; v++) {
        d[v] = CP[s][v];
        truoc[v] = s;
        final[v] = FALSE;
    }
```

```
    truoc[s] = 0; d[s] = 0; final[s] = TRUE;
    while(!final[t]) {
        minp = 2000;
        for(v = 1; v <= n; v++) {
            if((!final[v]) && (minp>d[v])) {
                u = v;
                minp = d[v];
            }
        }
        final[u] = TRUE;                    // u là đỉnh có nhãn tạm thời nhỏ nhất
        if(!final[t]) {
            for(v = 1; v <= n; v++) {
                if((!final[v]) && (d[u]+ CP[u][v] < d[v])) {
                    d[v] = d[u]+CP[u][v];
                    truoc[v] = u;
                }
            }
        }

    }
}
void main(void) {
    clrscr(); Init();
    Dijkstra();
    Result();
    getch();
}
```

6.7.3. Thuật toán Floy

Để tìm đường đi ngắn nhất giữa tất cả các cặp đỉnh của đồ thị, chúng ta có thể sử dụng n lần thuật toán ford_Bellman hoặc Dijkstra (trong trường hợp trọng số không âm). Tuy nhiên, trong cả hai thuật toán được sử dụng đều có độ phức tạp tính toán lớn (chí ít là $O(n^3)$). Trong trường hợp tổng quát, người ta thường dùng thuật toán Floy được mô tả như sau:

```
Procedure Floy;
(* Tìm đường đi ngắn nhất giữa tất cả các cặp đỉnh
Input:    Đồ thị cho bởi ma trận trọng số a[i, j], i, j = 1, 2,..., n.
Output:   Ma trận đường đi ngắn nhất giữa các cặp đỉnh d[i, j],
          i, j = 1, 2,..., n;
          d[i, j] là độ dài ngắn nhất từ i đến j.
          Ma trận ghi nhận đường đi p[i, j], i, j = 1, 2,..., n
          p[i, j] ghi nhận đỉnh đi trước đỉnh j trong đường đi ngắn nhất; *)
begin
   (* Bước khởi tạo *)
   for i:= 1 to n do
      for j := 1 to n do
      begin
         d[i, j] := a[i, j];
         p[i, j] := i;
      end;
   (* Bước lặp *)
   for k:= 1 to n do
      for i:= 1 to n do
         for j := 1 to n do
            if(d[i, j] > d[i, k] + d[k, j]) then
```

```
                begin
                    d[i, j] := d[i, k] + d[k, j];
                    p[i, j] := p[k, j];
                end;
        end;
```

Chương trình cài đặt thuật toán Foly tìm đường đi ngắn nhất giữa tất cả các cặp đỉnh của đồ thị được thể hiện như sau:

```
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
#include <math.h>
#include <dos.h>
#define MAX 10000
#define TRUE 1
#define FALSE 0
int A[50][50], D[50][50], S[50][50];
int n, u, v, k; FILE *fp;
void Init(void) {
    int i, j, k;
    fp = fopen("FLOY1.IN", "r");
    if(fp == NULL) {
        printf("\n Không có tệp input");
        getch(); return;
    }
    for(i = 1; i <= n; i++)
        for(j = 1; j <= n; j++)
            A[i][j] = 0;
    fscanf(fp, "%d%d%d", &n, &u, &v);
```

```
    printf("\n Số đỉnh đồ thị:%d", n);
    printf("\n Đi từ đỉnh:%d đến đỉnh %d:", u, v);
    printf("\n Ma trận trọng số:");
    for(i = 1; i <= n; i++) {
        printf("\n");
        for(j = 1; j <= n; j++) {
            fscanf(fp, "%d", &A[i][j]);
            printf("%5d", A[i][j]);
            if(i != j && A[i][j] == 0)
                A[i][j] = MAX;
        }
    }
    fclose(fp); getch();
}
void Result(void) {
    if(D[u][v]> = MAX) {
        printf("\n Không có đường đi");
        getch(); return;
    }
    else {
        printf("\n Đường đi ngắn nhất:%d", D[u][v]);
        printf("\n Đỉnh %3d", u);
        while(u != v) {
            printf("%3d", S[u][v]);
            u = S[u][v];
        }
    }
}
```

```
void Floy(void) {
    int i, j, k, found;
    for(i = 1; i <= n; i++) {
        for(j = 1; j <= n; j++) {
            D[i][j] = A[i][j];
            if(D[i][j] == MAX) S[i][j] = 0;
            else S[i][j] = j;
        }
    }
    /* Mảng D[i, j] là mảng chưa các giá trị khoảng cách ngắn nhất từ i đến j
    Mảng S[i, j] là mảng chứa giá trị phần tử ngay sau của i trên đường đi ngắn nhất
    từ i->j */
    for(k = 1; k <= n; k++) {
        for(i = 1; i <= n; i++) {
            for(j = 1; j <= n; j++) {
                if(D[i][k] != MAX && D[i][j]>(D[i][k]+D[k][j])) {
                // Tìm D[i, j] nhỏ nhất có thể có
                    D[i][j] = D[i][k]+D[k][j];
                    S[i][j] = S[i][k];
                // Ứng với nó là giá trị của phần tử ngay sau i
                }
            }
        }
    }
}
void main(void) {
    clrscr(); Init();
    Floy(); Result();
}
```

BÀI TẬP CHƯƠNG 6

6.1. Cho trước ma trận kề của đồ thị. Hãy viết chương trình tạo ra danh sách kề của đồ thị đó.

6.2. Cho trước danh sách kề của đồ thị, hãy tạo nên ma trận kề của đồ thị.

6.3. Một bàn cờ 8×8 được đánh số theo cách sau:

1	2	3	4	5	6	7	8
9	10	11	12	13	14	15	16
17	18	19	20	21	22	23	24
25	26	27	28	29	30	31	32
33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48
49	50	51	52	53	54	55	56
57	58	59	60	61	62	63	64

Mỗi ô có thể coi là một đỉnh của đồ thị. Hai đỉnh được coi là kề nhau nếu một con vua đặt ở ô này có thể nhảy sang ô kia sau một bước đi. Ví dụ: ô 1 kề với ô 2, 9, 10, ô 11 kề với 2, 3, 4, 10, 12, 18, 19, 20. Hãy viết chương trình tạo ma trận kề của đồ thị, kết quả in ra tệp king.out.

6.4. Bàn cờ 8 × 8 được đánh số như bài trên. Mỗi ô có thể coi là một đỉnh của đồ thị. Hai đỉnh được gọi là kề nhau nếu một con mã đặt ở ô này có thể nhảy sang ô kia sau một nước đi. Ví dụ ô 1 kề với 11, 18; ô 11 kề với 1, 5, 17, 21, 26, 28. Hãy viết chương trình lập ma trận kề của đồ thị, kết quả ghi vào tệp ma.out.

6.5. Hãy lập chương trình tìm một đường đi của con mã trên bàn cờ từ ô s đến ô t (s, t được nhập từ bàn phím).

6.6. Cho cơ sở dữ liệu ghi lại thông tin về N tuyến bay ($N \leq 100$) của một hãng hàng không. Trong đó, thông tin về mỗi tuyến bay được mô tả bởi: Điểm khởi hành (departure), điểm đến (destination), khoảng cách (length). Departure, destination là một xâu kí tự độ dài không quá 32, không chứa dấu trống ở giữa, length là một số nhỏ hơn 32767.

Ta gọi "***Hành trình bay***" từ điểm khởi hành A tới điểm đến B là dãy các hành trình $[A, A_1, n_1]$, $[A_1, A_2, n_2]$..., $[A_k, B, n_k]$ với A_i là điểm đến của tuyến i nhưng lại là điểm khởi hành của tuyến i +1, ni là khoảng cách của tuyến bay thứ i ($1 \leq i < k$). Trong đó, khoảng cách của hành trình là tổng khoảng cách của các tuyến mà hành trình đi qua ($n_1 + n_2 + ... + n_k$).

Cho tệp dữ liệu kiểu text hanhtrinh.in được ghi theo từng dòng, số các dòng trong tệp dữ liệu không vượt quá N, trên mỗi dòng ghi lại thông tin về một tuyến bay, trong đó departure, destination, length được phân biệt với nhau bởi một hoặc vài dấu trống. Hãy tìm giải pháp để thoả mãn nhu cầu của khách hàng đi từ A đến B theo một số tình huống sau:

Tìm hành trình có khoảng cách bé nhất từ A đến B. In ra màn hình từng điểm mà hành trình đã qua và khoảng cách của hành trình. Nếu hành trình không tồn tại hãy đưa ra thông báo "Hành trình không tồn tại".

Ví dụ về cơ sở dữ liệu hanhtrinh.in

New–York	Chicago	1000
Chicago	Denver	1000
New–York	Toronto	800

New_York	Denver	1900
Toronto	Calgary	1500
Toronto	Los_Angeles	1800
Toronto	Chicago	500
Denver	Urbana	1000
Denver	Houston	1500
Houston	Los_Angeles	1500
Denver	Los_Angeles	1000

Với điểm đi: New_York, điểm đến: Los_Angeles; chúng ta sẽ có kết quả sau:

Hành trình ngắn nhất:

New_York to Toronto to Los_Angeles; Khoảng cách: 2600.

6.7. Kế tục thành công với khối lập phương thần bí, Rubik sáng tạo ra dạng phẳng của trò chơi này gọi là trò chơi các ô vuông thần bí. Đó là một bảng gồm 8 ô vuông bằng nhau như hình 1. Chúng ta qui định trên mỗi ô vuông có một màu khác nhau. Các màu được kí hiệu bởi 8 số nguyên tương ứng với 8 màu cơ bản của màn hình EGA, VGA như hình 1. Trạng thái của bảng các màu được cho bởi dãy kí hiệu màu các ô được viết lần lượt theo chiều kim đồng hồ bắt đầu từ ô góc trên bên trái và kết thúc ở ô góc dưới bên trái. Ví dụ: trạng thái trong hình 1 được cho bởi dãy các màu tương ứng với dãy số (1, 2, 3, 4, 5, 6, 7, 8). Trạng thái này được gọi là trạng thái khởi đầu.

Biết rằng chỉ cần sử dụng 3 phép biến đổi cơ bản có tên là 'A', 'B', 'C' dưới đây bao giờ cũng chuyển được từ trạng thái khởi đầu về trạng thái bất kỳ:

'A': đổi chỗ dòng trên xuống dòng dưới. Ví dụ sau phép biến đổi A, hình 1 sẽ trở thành hình 2:

'B': thực hiện một phép hoán vị vòng quanh từ trái sang phải trên từng dòng. Ví dụ sau phép biến đổi B hình 1 sẽ trở thành hình 3:

'C': quay theo chiều kim đồng hồ bốn ô ở giữa. Ví dụ sau phép biến đổi C hình 1 trở thành hình 4:

Hình 1

1	2	3	4
8	7	6	5

Hình 2

8	7	6	5
1	2	3	4

Hình 3

1	1	2	3
5	8	7	6

Hình 4

1	7	2	4
6	6	3	5

Cho tệp dữ liệu Input.txt ghi lại 8 số nguyên trên một dòng, mỗi số được phân biệt với nhau bởi một dấu trống ghi lại trạng thái đích. Hãy tìm dãy các phép biến đổi sơ bản để đưa trạng thái khởi đầu về trạng thái đích sao cho số các phép biến đổi là ít nhất có thể được.

Dữ liệu ra được ghi lại trong tệp Output.txt, dòng đầu tiên ghi lại số các phép biến đổi, những dòng tiếp theo ghi lại tên của các thao tác cơ bản đã thực hiện, mỗi thao tác cơ bản được viết trên một dòng.

Bạn sẽ được thêm 20 điểm nếu sử dụng bảng màu thích hợp của màn hình để mô tả lại các phép biến đổi trạng thái của trò chơi. Ví dụ với trạng thái đích dưới đây sẽ cho ta kết quả như sau:

Input.txt	Output.txt
2 6 8 4 5 7 3 1	7
	B
	C
	A
	B
	C
	C
	B

6.8. Cho một mạng thông tin gồm N nút. Trong đó, đường truyền tin hai chiều trực tiếp từ nút i đến nút j có chi phí truyền thông tương ứng là một số nguyên A[i, j] = A[j, i], với A[i, j] ≥ 0, i ≠ j. Nếu đường truyền tin từ nút i_1 đến nút i_k phải thông qua các nút i_2,.. i_{k-1} thì chi phí truyền thông được tính bằng tổng các chi phí truyền thông A[i_1, i_2], A[i_2, i_3],... A[i_{k-1}, i_k]. Cho trước hai nút i và j. Hãy tìm một đường truyền tin từ nút i đến nút j sao cho chi phí truyền thông là thấp nhất.

Dữ liệu vào được cho bởi tệp TEXT có tên INP.NN. Trong đó, dòng thứ nhất ghi ba số N, i, j, dòng thứ k + 1 ghi k-1 số A[k, 1], A[k, 2],.., A[k, k-1], 1 ≤ k ≤ N.

Kết quả thông báo ra tệp TEXT có tên OUT.NN. Trong đó, dòng thứ nhất ghi chi phí truyền thông thấp nhất từ nút i đến nút j, dòng thứ 2 ghi lần lượt các nút trên đường truyền tin có chi phí truyền thông thấp nhất từ nút i tới nút j.

6.9. Cho một mạng thông tin gồm N nút. Trong đó, đường truyền tin hai chiều trực tiếp từ nút i đến nút j có chi phí truyền thông tương ứng là một số nguyên A[i, j] = A[j, i], với A[i, j] ≥ 0, i ≠ j. Nếu đường truyền tin từ nút i_1 đến nút i_k phải thông qua các nút i_2,.. i_{k-1} thì chi phí truyền thông được tính bằng tổng các chi phí truyền thông A[i_1, i_2], A[i_2, i_3],... A[i_{k-1}, i_k]. Biết rằng, giữa hai nút bất kỳ của mạng thông tin đều tồn tại ít nhất một đường truyền tin.

Để tiết kiệm đường truyền, người ta tìm cách loại bỏ đi một số đường truyền tin mà vẫn đảm bảo được tính liên thông của mạng. Hãy tìm một phương án loại bỏ đi những đường truyền tin, sao cho ta nhận được một mạng liên thông có chi phí tối thiểu nhất có thể được.

Dữ liệu vào được cho bởi tệp TEXT có tên INP.NN. Trong đó, dòng thứ nhất ghi số N, dòng thứ k + 1 ghi k-1 số A[k, 1], A[k, 2],.., A[k, k-1], 1 ≤ k ≤ N.

Kết quả thông báo ra tệp TEXT có tên OUT.NN trong đó dòng thứ nhất ghi chi phí truyền thông nhỏ nhất trong toàn mạng. Từ dòng thứ 2 ghi lần lượt các nút trên đường truyền tin, mỗi đường truyền ghi trên một dòng.

6.10. Cho tệp dữ liệu được tổ chức giống như bài 6.6. Hãy tìm tất cả các hành trình đi từ điểm s đến t.

6.11. Cho tệp dữ liệu được tổ chức giống như bài 6.6. Hãy tìm hành trình đi từ điểm s đến t sao cho hành trình đi qua nhiều node nhất.

6.12. Cho tệp dữ liệu được tổ chức giống như bài 6.6. Hãy tìm hành trình đi từ điểm s đến t sao cho hành trình đi qua ít node nhất.

6.13. Tìm hiểu thuật toán leo đồi trên đồ thị và ứng dụng của nó trong lĩnh vực trí tuệ nhân tạo.

CHƯƠNG 7

SẮP XẾP VÀ TÌM KIẾM (SORTING AND SEARCHING)

7.1. ĐẶT BÀI TOÁN

Sắp xếp là quá trình bố trí lại các phần tử của một tập đối tượng nào đó theo một thứ tự ấn định tăng dần (increasing), hoặc giảm dần (decreasing). Bài toán sắp xếp xuất hiện trong bất kỳ lĩnh vực nào của tin học, phục vụ những ứng dụng riêng của hệ thống, từ những ứng dụng ẩn bên trong của hệ điều hành như bài toán điều khiển quá trình (Proccess Control Problem), bài toán lập lịch cho CPU (CPU Schedulling), bài toán quản lý bộ nhớ (Memory Management)... cho tới những ứng dụng thông thường như sắp xếp dãy số, sắp xếp các từ, các câu, các bản ghi theo thứ tự đều có liên quan tới quá trình sắp xếp.

Tập đối tượng cần được sắp xếp có thể xuất hiện dưới nhiều dạng khác nhau, các đối tượng đó có thể là các đối tượng dữ liệu kiểu cơ bản như sắp xếp dãy số, sắp xếp kí tự, sắp xếp string hoặc là các đối tượng tổng quát như một cấu trúc bao gồm một số trường thông tin phản ánh đối tượng. Chúng ta qui ước đối tượng cần được sắp xếp là các cấu trúc, và quá trình sắp xếp được thực hiện trên một trường nào đó gọi là trường khoá.

Có nhiều thuật toán sắp xếp khác nhau để sắp xếp các đối tượng. Tuy nhiên, để lựa chọn một thuật toán sắp xếp tốt, chúng ta cần đánh giá thuật toán theo các hai khía cạnh: đó là sự chiếm dụng bộ nhớ khi áp dụng giải thuật và thời gian thực hiện giải thuật. Đối với thời gian thực hiện giải thuật, chúng ta cũng cần đánh giá chi phí thời gian trong trường hợp tốt nhất, trung bình và xấu nhất đối với nguồn dữ liệu vào.

Chúng ta cũng chỉ đưa ra những kỹ thuật lập trình, thông qua giải thuật và kết quả đánh giá thuật toán mà không chứng minh lại những kết quả đó, vì nó đã được trình bày trong một chuyên đề khác của tin học.

Những thuật toán sắp xếp và tìm kiếm sẽ được bàn luận trong chương này bao gồm các thuật toán sắp xếp đơn giản như: chọn trực tiếp (Selection), thuật toán sủi bọt (Bubble), thuật toán chèn trực tiếp (Insertion), các thuật toán sắp xếp nhanh như quick sort, merge sort, heap sort. Trong tất cả các ví dụ minh họa cho giải thuật sắp xếp và tìm kiếm, chúng ta sẽ sử dụng tập các số nguyên dưới đây làm ví dụ sắp xếp. Dãy số nguyên này sẽ không được nhắc lại trong khi giải thích mỗi thuật toán sắp xếp.

42 23 74 11 65 58 94 36 99 87

7.2. GIẢI THUẬT SELECTION SORT

Nội dung của Selection Sort là lần lượt chọn phần tử nhỏ nhất trong dãy chỉ số $k_1, k_2, \ldots, k_n$ với $i = 0, 1, \ldots, n$; $k_i < k_{i+1} < \ldots, k_n$ và đổi chỗ cho phần tử thứ k_i. Như vậy, sau $j = n - 1$ lần chọn, chúng ta sẽ só dãy khoá được sắp xếp theo thứ tự tăng dần. Đối với dãy số trên, chúng ta sẽ thực hiện như sau:

- Lần chọn thứ 0: Tìm trong khoảng từ 0 đến n - 1 bằng cách thực hiện n - 1 lần so sánh để xác định phần tử min_0 và đổi chỗ cho phần tử ở vị trí 0.
- Lần chọn thứ 1: Tìm trong khoảng từ 1 đến n - 1 bằng cách thực hiện n - 2 lần so sánh để xác định phần tử min_1 và đổi chỗ cho phần tử ở vị trí 1.
- Thực hiện tiếp các lần chọn thứ 2, 3 ...
- Lần chọn thứ i: Tìm trong khoảng từ i đến n - 1 bằng cách thực hiện n - i lần so sánh để xác định phần tử min_i và đổi chỗ cho phần tử ở vị trí i.

- Thực hiện tiếp các lần chọn thứ i + 1, i + 2, ...
- Lần chọn thứ n - 2: Tìm trong khoảng từ n - 2 đến n - 1 bằng cách thực hiện 1 lần so sánh để xác định phần tử min_{n-2} và đổi chỗ cho phần tử ở vị trí n - 2.

Độ phức tạp tính toán của giải thuật Selection Sort là:

$C_{min} = C_{max} = C_{tb} = n(n - 1)/2$

Quá trình sắp xếp dãy số được minh họa thông qua bảng sau:

i	ki	1	2	3	4	5	6	7	8	9
0	42	11	11	11	11	11	11	11	11	11
1	23	23	23	23	23	23	23	23	23	23
2	74	74	74	36	36	36	36	36	36	36
3	11	42	42	42	42	42	42	42	42	42
4	65	65	65	65	65	58	58	58	58	58
5	58	58	58	58	58	65	65	65	65	65
6	94	94	94	94	94	94	74	74	74	74
7	36	36	36	74	74	74	94	87	87	87
8	99	99	99	99	99	99	99	99	94	94
9	87	87	87	87	87	87	87	94	99	99

Chương trình được cài đặt như sau:

```
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
#include <alloc.h>
#include <dos.h>
void Select(int *, int);
void Init(int *, int);
void In(int *, int);
void Init(int *A, int n) {
   int i;
```

```
    printf("\n Tạo lập dãy số:");
    for(i = 0; i < n; i ++) {
        A[i] = random(1000);
        printf("%5d",A[i]);
    }
    delay(1000);
}
void Select(int *A, int n) {
    register i,j,temp;
    for(i = 0; i < n - 1; i ++) {
        for(j = i + 1; j < n; j ++) {
            if(A[i] > A[j]) {
                temp = A[i];
                A[i] = A[j];
                A[j] = temp;
            }
        }
        In(A,n);
    }
}
void In(int *A, int n) {
    register int i;
    for(i = 0; i < n; i ++)
        printf("%5d",A[i]);
    delay(1000);
}
```

```
void main(void) {
   int *A,n; clrscr();
   printf("\n Nhập n = "); scanf("%d",&n);
   A = (int *) malloc(n*sizeof(int));
   Init(A,n); Select(A,n);
   free(A);
}
```

7.3. GIẢI THUẬT INSERTION SORT

Giải thuật Insert Sort được thực hiện dựa trên kinh nghiệm của những người chơi bài. Khi trên tay có i - 1 lá bài đã được sắp xếp đang ở trên tay, nay ta thêm lá bài thứ i thì lá bài đó được so sánh với lá bài i - 1, i - 2, ... để tìm được vị trí thích hợp và chèn vào quân bài thứ i.

Với nguyên tắc sắp bài như vậy, giải thuật được thực hiện như sau:

- Lấy phần tử đầu tiên i_0, đương nhiên tập một phần tử là tập đã được sắp xếp.
- Lấy phần tử thứ i_1 chọn vị trí thích hợp của phần tử thứ i_1 trong tập hai phần tử và thực hiện đổi chỗ.
- Thực hiện tiếp lấy phần tử thứ i_2, i_3 ...
- Lấy tiếp phần tử thứ i_k chọn vị trí thích hợp của phần tử thứ i_k trong tập hai i_{k-1} phần tử và thực hiện đổi chỗ, dãy sẽ được sắp xếp hoàn toàn sau n - 1 lần chèn phần tử vào vị trí thích hợp.

  Độ phức tạp bé nhất của thuật toán là: $C_{min} = (n - 1)$;

  Độ phức tạp lớn nhất của thuật toán là: $n(n - 1)/2 = O(n^2)$

  Độ phức tạp trung bình của thuật toán là: $(n^2 + n - 2)/4 = O(n^2)$

  Quá trình sắp xếp theo Insertion Sort được mô tả như sau:

Lượt	1	2	3	4	...	8	9	10
Khoá	42	23	74	11	...	36	99	87
1	42	23	23	11	...	11	11	11
2		42	42	23	...	23	23	23
3			74	42	...	42	36	36
4				74	...	58	42	42
5					...	65	58	58
6					...	74	65	65
7					...	94	74	74
8					...		94	87
9					...		99	95
10					...			99

Thuật toán được cài đặt như sau:

```
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
#include <alloc.h>
#include <dos.h>
void Insert(int *, int);
void Init(int *, int);
void In(int *, int);
void Init(int *A, int n) {
    int i;
    printf("\n Tạo lập dãy số:");
    for(i = 0; i < n; i ++) {
        A[i] = random(1000);
        printf("%5d",A[i]);
    }
```

```
    delay(1000);
}
void Insert(int *A, int n) {
    register i,j,temp;
    for(i = 1; i < n; i ++) {
        temp = A[i];
        for(j = i - 1; j > = 0 && temp < A[j]; j --)
            A[j + 1] = A[j];
        A[j + 1] = temp;
        printf("\n");
        In(A,i + 1);
    }
}
void In(int *A, int n) {
    register int i;
    for(i = 0; i < n; i ++)
        printf("%5d",A[i]);
    delay(1000);
}
void main(void) {
    int *A,n; clrscr();
    printf("\n Nhập n = "); scanf("%d",&n);
    A = (int *) malloc(n*sizeof(int));
    Init(A,n); Insert(A,n);
    free(A);
}
```

7.4. GIẢI THUẬT BUBBLE SORT

Giải thuật Bubble Sort được thực hiện bằng cách đổi chỗ liên tiếp hai phần tử kế cận khi chúng ngược thứ tự. Quá trình thực hiện được duyệt từ đáy lên đỉnh. Như vậy, sau lần duyệt thứ nhất, phần tử lớn nhất sẽ được xếp đúng ở vị trí thứ n - 1, ở lần duyệt thứ k thì k phần tử lớn nhất đã được xếp đúng vị trí n - 1, n - 2, ..., n - k + 1. Sau lần duyệt thứ n - 1, toàn bộ n phần tử sẽ được sắp xếp. Với phương pháp này, các phần tử có giá trị nhỏ được nổi dần lên như nước sủi bọt nhờ đó nó có tên gọi "phương pháp sủi bọt".

Độ phức tạp của thuật toán Bubble Sort là:

$C_{min} = C_{max} = C_{tb} = n(n - 1)/2$.

Chương trình mô tả thuật toán Bubble Sort được cài đặt như sau:

```
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
#include <alloc.h>
#include <dos.h>
void Bubble(int *, int);
void Init(int *, int);
void In(int *, int);
void Init(int *A, int n) {
   int i;
   printf("\n Tạo lập dãy số:");
   for(i = 0; i < n; i ++) {
      A[i] = random(1000);
      printf("%5d",A[i]);
   }
```

```
    delay(1000);
}
void Bubble(int *A, int n) {
    register i,j,temp;
    for(i = 1; i < n; i ++) {
        for(j = n - 1; j > = i; j --) {
            if(A[j - 1] > A[j]) {
                temp = A[j - 1];
                A[j - 1] = A[j];
                A[j] = temp;
            }
        }
        printf("\n Kết quả lần:%d", i);
        In(A,n);
    }
}
void In(int *A, int n) {
    register int i;
    for(i = 0; i < n; i ++)
        printf("%5d",A[i]);
    delay(1000);
}
void main(void) {
    int *A,n; clrscr();
    printf("\n Nhập n = "); scanf("%d",&n);
    A = (int *) malloc(n*sizeof(int));
```

```
    Init(A,n); Bubble(A,n);
    free(A);
}
```

7.5. GIẢI THUẬT SHAKER SORT

Thuật toán Shaker Sort là cải tiến của thuật toán Bubble Sort. Trong đó, sau mỗi lần duyệt đi để xếp đúng vị trí phần tử lớn nhất, chúng ta thực hiện duyệt lại để sắp đúng vị trí phần tử nhỏ nhất. Dãy sẽ được sắp sau [n/2] + 1 lần duyệt. Chương trình mô tả thuật toán Shaker Sort được thực hiện như sau:

```
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
#include <alloc.h>
#include <dos.h>
void Shaker(int *, int);
void Init(int *, int);
void In(int *, int);
void Init(int *A, int n) {
    int i;
    printf("\n Tạo lập dãy số:");
    for(i = 0; i < n; i ++) {
        A[i] = random(1000);
        printf("%5d",A[i]);
    }
    delay(1000);
}
void Shaker(int *A, int n) {
    register i,j,temp, exchange;
```

```
  do {
     exchange = 0;
     for(i = n - 1; i > 0; i --) {
        if(A[i - 1] > A[i]) {
           temp = A[i - 1];
           A[i - 1] = A[i];
           A[i] = temp;
           exchange = 1;
        }
     }
     for(j = 1; j < n; j ++) {
        if(A[j - 1] > A[j]) {
           temp = A[j - 1];
           A[j - 1] = A[j];
           A[j] = temp;
           exchange = 1;
        }
     }
     printf("\n Kết quả lần:");
     In(A,n);
  }while(exchange);
}
void In(int *A, int n) {
   register int i;
   for(i = 0; i < n; i ++)
      printf("%5d",A[i]);
```

```
    delay(1000);
}
void main(void) {
    int *A,n; clrscr();
    printf("\n Nhập n = "); scanf("%d",&n);
    A = (int *) malloc(n*sizeof(int));
    Init(A,n); Shaker(A,n);
    free(A);
}
```

7.6. GIẢI THUẬT QUICK SORT

Phương pháp sắp xếp kiểu phân đoạn là một cải tiến của phương pháp Selection Sort. Đây là một phương pháp tốt do C.A.R. Hoare đưa ra và đặt tên cho nó là giải thuật Quick Sort.

Nội dung chủ đạo của phương pháp này là chọn ngẫu nhiên một phần tử nào đó của dãy làm khoá chốt. Tính từ khoá chốt, các phần tử nhỏ hơn khoá phải được xếp vào trước chốt (đầu dãy), mọi phần tử sau chốt được xếp vào sau chốt (cuối dãy). Để làm được việc đó, các phần tử trong dãy sẽ được so sánh với khoá chốt và tráo đổi vị trí cho nhau, hoặc cho khoá chốt nếu phần tử đó lớn hơn chốt mà lại nằm trước chốt hoặc nhỏ hơn chốt nhưng lại nằm sau chốt. Khi việc đổi chỗ lần đầu tiên đã thực hiện xong thì dãy hình thành hai đoạn: một đoạn bao gồm các phần tử nhỏ hơn chốt, một đoạn gồm các phần tử lớn hơn chốt, còn chốt chính là vị trí của phần tử trong dãy được sắp xếp.

Áp dụng kỹ thuật như trên cho mỗi đoạn trước chốt và sau chốt cho tới khi các đoạn còn lại hai phần tử thì việc ghi nhớ không còn cần thiết nữa. Dãy sẽ được sắp xếp khi tất cả các đoạn được xử lý xong. Ví dụ với dãy :

42 23 74 11 65 58 94 36 99 87

Ta chọn chốt đầu tiên là 42. Để phát hiện ra hai khoá cần đổi chỗ cho nhau, ta dùng hai biến i, j với giá trị ban đầu i = 2, j = 10. Nếu $k_i < 42$ thì tiếp tục tăng i và lặp lại cho tới khi gặp phần tử thứ $k_i > 42$. Duyệt các phần tử thứ k_j với 42 nếu $k_j > 42$ thì j giảm đi một, cho tới khi gặp phần tử thứ $k_j < 42$ thì phần tử thứ k_i và k_j được đổi chỗ cho nhau. Quá trình sẽ được lặp lại với k_i và k_j cho tới khi i = j chính là vị trí dành cho khoá 42. Cuối cùng chúng ta đổi chỗ 42 cho khoá cho k_j.

42 23 74 11 65 58 94 36 99 87

42 23 74 11 65 58 94 36 99 87

42 23 36 11 65 58 94 74 99 87

42 23 36 11 65 58 94 74 99 87

42 23 36 11 65 58 94 74 99 87 (i > j)

11 23 36 42 65 58 94 74 99 87

Như vậy, kết thúc lần thứ nhất, chúng ta được hai đoạn được phân biệt bởi khoá 42 như sau:

(11 23 36) [42] (65 58 94 74 99 87)

Quá trình được lặp lại tương tự cho từng phân đoạn cho tới khi dãy được sắp xếp hoàn toàn. Chúng ta có thể cài đặt giải thuật bằng việc sử dụng stack hoặc đệ qui.

Độ phức tạp tính toán của giải thuật Quick Sort:

Trường hợp tốt nhất $C_{max} = C_{tb} = O(n \log_2 n)$

Trường hợp xấu nhất $C_{min} = k.O(n^2)$

Sau đây là chương trình cài đặt giải thuật Quick Sort bằng phương pháp đệ qui.

```
#include <stdio.h>

#include <conio.h>
```

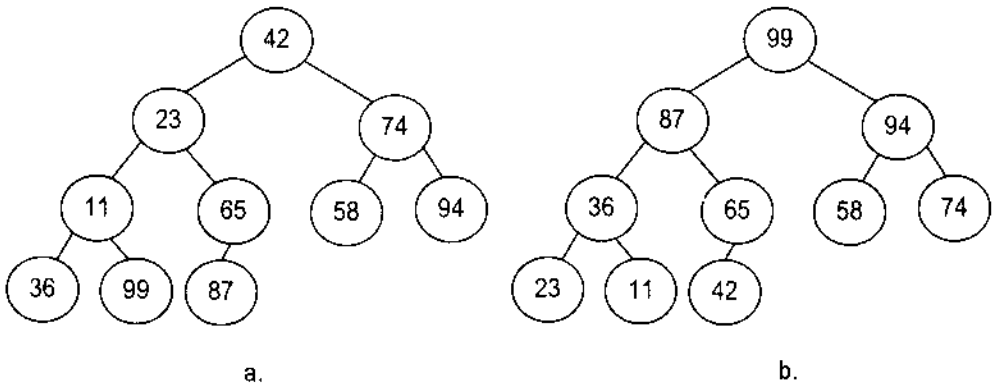
```
#include <stdlib.h>
#include <alloc.h>
#include <dos.h>
void qs(int *, int ,int);
void Quick(int *,int);
void Init(int *, int);
void In(int *, int);
void Init(int *A, int n) {
   int i;
   printf("\n Tạo lập dãy số:");
   for(i = 0; i < n; i ++) {
      A[i] = random(1000);
      printf("%5d",A[i]);
   }
   delay(1000);
}
void Quick(int *A, int n) {
   qs(A,0,n - 1);
}
void qs(int *A, int left,int right) {
   register i,j; int x,y;
   i = left; j = right;
   x = A[(left + right)/2];
   do {
      while(A[i] < x && i < right) i ++ ;
      while(A[j] > x && j > left) j -- ;
```

```
        if(i < = j) {
            y = A[i]; A[i] = A[j]; A[j] = y;
            i ++ ; j -- ;
        }
    } while (i < = j);
    if(left < j) qs(A,left,j);
    if(i < right) qs(A,i,right);
}
void In(int *A, int n) {
    register int i;
    for(i = 0; i < n; i ++)
        printf("%5d",A[i]);
    delay(1000);
}
void main(void) {
    int *A,n; clrscr();
    printf("\n Nhập n = "); scanf("%d",&n);
    A = (int *)malloc(n*sizeof(int));
    Init(A,n); Quick(A,n); printf("\n");
    In(A,n); getch();
    free(A);
}
```

7.7. GIẢI THUẬT HEAP SORT

Heap là một cây nhị phân được biểu diễn bằng một mảng, mảng đó biểu diễn một cây nhị phân hoàn chỉnh sao cho khóa ở node cha bao giờ cũng lớn hơn khoá của node con của nó.

Sắp xếp kiểu Heap Sort được tiến hành qua hai giai đoạn. Giai đoạn đầu tiên cây nhị phân biểu diễn bảng khoá được biến đổi để đưa về một heap. Như vậy, đối với heap, nếu j là chỉ số của node con thì [j/2] là chỉ số của node cha. Theo định nghĩa của heap thì node con bao giờ cũng nhỏ hơn node cha. Như vậy, node gốc của heap là khóa có giá trị lớn nhất trong mọi node. Ví dụ cây ban đầu là cây 7.1a thì heap của nó là 7.1b.



Hình 7.1 Tìm kiếm theo giải thuật Heap Sort

Để chuyển cây nhị phân 7.1a thành cây nhị phân 7.1b là một heap, chúng ta thực hiện duyệt từ dưới lên (bottom up). Node lá đương nhiên là một heap. Nếu cây con bên trái và cây con bên phải đều là một heap thì toàn bộ cây cũng là một heap. Như vậy, để tạo thành heap, chúng ta thực hiện so sánh nội dung node bên trái, nội dung node bên phải với node cha của nó, node nào có giá trị lớn hơn sẽ được thay đổi làm nội dung của node cha. Quá trình lần ngược lại cho tới khi gặp node gốc, khi đó nội dung node gốc chính là khoá có giá trị lớn nhất.

Giai đoạn thứ hai của giải thuật là đưa nội dung của node gốc về vị trí cuối cùng và nội dung của node cuối cùng được thay vào vị trí node gốc, sau đó coi như node cuối cùng như đã bị loại bỏ vì thực tế node cuối cùng là giá trị lớn nhất trong dãy số.

Cây mới được tạo ra (không kể phần tử loại bỏ) không phải là một heap, chúng ta lại thực hiện vun thành đống và thực hiện tương tự như trên cho tới khi đống còn một phần tử là phần tử bé nhất của dãy.

Độ phức tạp thuật toán của Heap Sort

$C_{max} = C_{tb} = O(n \log_2 n)$

Giải thuật Heap Sort được cài đặt như sau:

```
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
#include <alloc.h>
#include <dos.h>
void Heap(int *, int);
void Init(int *, int);
void In(int *, int);
void Init(int *A, int n) {
   int i;
   printf("\n Tạo lập dãy số:");
   for(i = 0; i < n; i ++) {
      A[i] = random(1000);
      printf("%5d",A[i]);
   }
   delay(1000);
}
void Heap(int *A, int n) {
   int k,x,s,f,ivalue;
   for(k = 1; k < n; k ++) {
```

```
        x = A[k];
        s = k; f = (s - 1)/2;
        while(s > 0 && A[f] < x) {
            A[s] = A[f];
            s = f; f = (s - 1)/2;
        }
        A[s] = x;
    }
    for(k = n - 1; k > 0; k --) {
        ivalue = A[k];
        A[k] = A[0];
        f = 0;
        if(k == 1)
            s = - 1;
        else
            s = 1;
        if(k > 2 && A[2] > A[1])
            s = 2;
        while(s >= 0 && ivalue < A[s]) {
            A[f] = A[s];
            f = s; s = 2*f + 1;
            if(s + 1 < = k - 1 && A[s] < A[s + 1])
                s = s + 1;
            if(s > k - 1)
                s = - 1;
        }
```

```
        A[f] = ivalue;
    }
}
void In(int *A, int n) {
    register int i;
    for(i = 0; i < n; i ++)
        printf("%5d",A[i]);
    delay(1000);
}
void main(void) {
    int *A,n; clrscr();
    printf("\n Nhập n = "); scanf("%d",&n);
    A = (int *) malloc(n*sizeof(int));
    Init(A,n); Heap(A,n); printf("\n");
    In(A,n); getch();
    free(A);
}
```

7.8. GIẢI THUẬT MERGE SORT

Sắp xếp theo Merge Sort là phương pháp sắp xếp bằng cách trộn hai danh sách đã được sắp xếp thành một danh sách đã được sắp xếp. Phương pháp Merge Sort được tiến hành thông qua các bước sau:

- Bước 1: Coi danh sách là n danh sách con mỗi danh sách con gồm một phần tử, như vậy các danh sách con đã được sắp xếp. Trộn từng cặp hai danh sách con kế cận thành một danh sách có hai phần tử đã được sắp xếp, chúng ta nhận được n/2 danh sách con đã được sắp xếp.

- Bước 2: Xem danh sách cần sắp xếp như n/2 danh sách con đã được sắp xếp. Trộn cặp hai danh sách kế cận thành từng danh sách có 4 phần tử đã được sắp xếp, chúng ta nhận được n/4 danh sách con.
- Thực hiện tiếp các bước 3, 4 ...
- Bước thứ i: Làm tương tự như bước i - 1. Quá trình được tiếp tục khi chúng ta nhận được danh sách có n phần tử đã được sắp xếp. Ví dụ với dãy:

  42 23 74 11 68 58 94 36

  Lần 1: [23 42] [11 74] [58 68] [94 36]

  Lần 2: [11 23 42 74] [36 58 68 94]

  Lần 3: [11 23 42 36 58 68 74 94]

Chương trình cài đặt giải thuật Merge Sort được thực hiện như sau:

```
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
#include <alloc.h>
#include <dos.h>
#define MAX 10
void Merge(int *, int);
void Init(int *, int);
void In(int *, int);
void Init(int *A, int n) {
    int i;
    printf("\n Tạo lập dãy số:");
    for(i = 0; i < n; i ++) {
```

```
        A[i] = random(1000);
        printf("%5d",A[i]);
    }
    delay(1000);
}
void Merge(int *A, int n) {
    int i,j,k,low1,up1,low2,up2,size;
    int *dstam; size = 1; dstam = (int *) malloc(n*sizeof(int));
    while(size < n) {
        low1 = 0; k = 0;
        while(low1 + size < n) {
            low2 = low1 + size;  up1 = low2 - 1;
            if(low2 + size - 1 < n)
                up2 = low2 + size - 1;
            else
                up2 = n - 1;
            for(i = low1, j = low2; i < = up1 && j < = up2; k ++) {
                if(A[i] < = A[j])
                    dstam[k] = A[i ++ ];
                else
                    dstam[k] = A[j ++ ];
            }
            for(; i < = up1; k ++)
                dstam[k] = A[i ++ ];
            for(; j < = up2; k ++)
                dstam[k] = A[j ++ ];
```

```
            low1 = up2 + 1;
        }
        for(i = low1; k < n; i ++)
            dstam[k ++ ] = A[i];
        for(i = 0; i < n; i ++)
            A[i] = dstam[i];
        size* = 2;
    }
    printf("\n Kết quả:");
    In(A,n); free(dstam);
}
void In(int *A, int n) {
    register int i;
    for(i = 0; i < n; i ++)
        printf("%5d",A[i]);
    delay(1000);
}
void main(void) {
    int *A,n; clrscr();
    printf("\n Nhập n = "); scanf("%d",&n);
    A = (int *) malloc(n*sizeof(int));
    Init(A,n); Merge(A,n); printf("\n");
    free(A);
}
```

7.9. TÌM KIẾM (SEARCHING)

Tìm kiếm là công việc quan trọng đối với các hệ thống tin học và có liên quan mật thiết với quá trình sắp xếp dữ liệu. Bài toán tìm kiếm tổng quát có thể được phát biểu như sau:

"Cho một bảng gồm n bản ghi R_1, R_2, ..., R_n. Với mỗi bản ghi R_i được tương ứng với một khoá k_i (trường thứ i trong record). Hãy tìm bản ghi có giá trị của khoá bằng X cho trước".

Nếu chúng ta tìm được bản ghi có giá trị khóa là X thì phép tìm kiếm được thoả (successful). Nếu không có giá trị khóa nào là X, thì quá trình tìm kiếm là không thoả (unsuccessful). Sau quá trình tìm kiếm, có thể xuất hiện yêu cầu bổ sung thêm bản ghi mới có giá trị khóa là X thì giải thuật được gọi là giải thuật tìm kiếm bổ sung.

7.9.1. Tìm kiếm tuần tự (Sequential Searching)

Tìm kiếm tuần tự là kỹ thuật tìm kiếm cổ điển trên một danh sách chưa được sắp xếp. Nội dung cơ bản của phương pháp tìm kiếm tuần tự là duyệt từ bản ghi thứ nhất cho tới bản ghi cuối cùng, và so sánh lần lượt giá trị của khoá với giá trị X cần tìm. Trong quá trình duyệt, nếu có bản ghi trùng với giá trị X thì chúng ta đưa ra vị trí của bản ghi trong dãy, nếu duyệt tới cuối dãy mà không có bản ghi nào có giá trị của khoá trùng với X thì quá trình tìm kiếm trả lại giá trị - 1 (- 1 được hiểu là giá trị khoá X không thuộc dãy). Chương trình cài đặt phương pháp tìm kiếm tuần tự được thực hiện như sau:

```
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
#include <alloc.h>
#include <dos.h>
int Sequential(int *, int, int);
```

```
void Init(int *, int);
void Init(int *A, int n) {
    int i;
    printf("\n Tạo lập dãy số:");
    for(i = 0; i < n; i ++) {
        A[i] = random(1000);
        printf("%5d",A[i]);
    }
    delay(1000);
}
int Bubble(int *A, int x, int n) {
    register i,temp;
    for(i = 0; i < n ; i ++) {
        if(A[i] == X)
            return(i);
    }
    return( -1);
}
void main(void) {
    int *A,n, x, k; clrscr();
    printf("\n Nhập n = "); scanf("%d",&n);
    printf("\n Số x cần tìm:"); scanf("%d", &x);
    A = (int *) malloc(n*sizeof(int));
    k = Sequential(A, x, n);
    if(k > = 0)
        printf("\n %d ở vị trí %d", x, k);
```

```
    else
        printf("\n %d không thuộc dãy");
    free(A); getch();
}
```

7.9.2. Tìm kiếm nhị phân (Binary Searching)

Tìm kiếm nhị phân là phương pháp tìm kiếm phổ biến được thực hiện trên một dãy đã được sắp thứ tự. Nội dung của giải thuật được thực hiện như sau: lấy khóa cần tìm kiếm X so sánh với nội dung của khóa của phần tử ở giữa, vị trí của phần tử ở giữa là *mid = (low + hight)/2*, trong đó cận dưới *low = 0*, cận trên *hight = n - 1*. Vì dãy đã được sắp xếp nên nếu nội dung của khóa tại vị trí giữa lớn hơn X thì phần tử cần tìm thuộc khoảng [mid + 1, hight], nếu nội dung của khóa tại vị trí giữa nhỏ hơn X thì phần tử cần tìm thuộc khoảng [low, mid - 1], nếu nội dung của khóa tại vị trí giữa trùng với X thì đó chính là phần tử cần tìm. Ở bước tiếp theo, nếu nội dung của khóa tại vị trí giữa lớn hơn X thì ta dịch chuyển cận dưới *low* lên vị trí *mid + 1*, nếu nội dung của khóa tại vị trí giữa nhỏ hơn X thì ta dịch chuyển cận trên về vị trí *mid - 1*. Quá trình được lặp lại cho tới khi gặp khóa có nội dung trùng với X hoặc cận dưới vượt quá cận trên hay X không thuộc dãy. Thuật toán tìm kiếm nhị phân được minh họa như sau:

```
int Binary_Search(int *A, int X, int n) {
    int mid, low = 0, hight = n - 1;
    while (low < = hight) {          // lặp nếu cận dưới vẫn nhỏ hơn cận trên
        mid = (low + hight) / 2;                 // xác định vị trí phần tử ở giữa
        if(X > A[mid])    low = mid + 1;         // X thuộc [mid + 1, hight]
        else if(X < A[mid]) hight = mid - 1;     // X thuộc [low, mid - 1]
        else   return(mid);
    }
```

```
        return( -1);                    // X không thuộc dãy
    }
```

Chương trình cụ thể được cài đặt như sau:

```
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
#include <alloc.h>
#include <dos.h>
int Binary_Search(int *, int, int);
void Bubble(int *, int);
void Init(int *, int);
int Binary_Search(int *A, int X, int n) {
   int mid, low = 0, hight = n - 1;
   while (low < = hight) {
      mid = (low + hight)/2;
      if(X > A[mid]) low = mid + 1;
      else if(X < A[mid]) hight = mid - 1;
      else  return (mid);
   }
   return( - 1);
}
void Init(int *A, int n) {
   int i;
   printf("\n Tạo lập dãy số:");
   for(i = 0; i < n; i ++) {
      A[i] = random(1000);
```

```
        printf("%5d",A[i]);
    }
    delay(1000);
}
void Bubble(int *A, int n) {
    register i,j,temp;
    for(i = 1; i < n; i ++) {
        for(j = n - 1; j > = i; j --) {
            if(A[j - 1] > A[j]) {
                temp = A[j - 1];
                A[j - 1] = A[j];
                A[j] = temp;
            }
        }
        printf("\n Kết quả lần:%d", i);
        In(A,n);
    }
}
void In(int *A, int n) {
    register int i;
    for(i = 0; i < n; i ++)
        printf("%5d",A[i]);
    delay(1000);
}
void main(void) {
    int *A,n, X, k; clrscr();
```

```
    printf("\n Nhập n = "); scanf("%d",&n);
    printf("\n Số cần tìm X = "); scanf("%d",&X);
    A = (int *) malloc(n*sizeof(int));
    Init(A,n); Bubble(A,n); k = Binary_Search(A, X, n);
    if(k > 0)
        printf ("\n %d ở vị trí số %d", X, k);
    else
        printf("\n %d không thuộc dãy");
    getch();
    free(A);
}
```

BÀI TẬP CHƯƠNG 7

7.1. Cài đặt chương trình theo thuật toán Quick Sort không dùng phương pháp đệ qui mà dùng cấu trúc stack.

7.2. Tìm hiểu về giải thuật Shell - Sort là phương pháp cải tiến của Insertion Sort.

7.3. Cài đặt lại giải thuật Bubble Sort sao cho các node nhỏ được đẩy dần về phía trước.

7.4. Một Ternary Heap là cây tam phân gần đầy được cài đặt bằng mảng một chiều, mỗi node có ba node con. Nội dung của node cha bao giờ cũng lớn hơn hoặc bằng nội dung của node con, các node được đánh số từ 0 đến n - 1, node i có 3 con là 3i + 1, 3i + 2, 3i + 3. Hãy cài đặt giải thuật Ternary Heap.

7.5. Cài đặt giải thuật Bubble Sort trên tệp.

7.6. Cài đặt giải thuật Insertion Sort trên tệp.

7.7. Cài đặt giải thuật Quick trên tệp.

7.8. Cài đặt các giải thuật sắp xếp theo nhiều khoá khác nhau.

7.9. Nghiên cứu và cài đặt thuật toán tìm kiếm tam phân.

7.10. Nghiên cứu và cài đặt thuật toán sắp xếp kiểu hoà nhập thực hiện trên tệp.

7.11. Viết chương trình chuyển đổi một tệp dữ liệu được tổ chức theo khuôn dạng *.DBF thành tệp kiểu text. Ngược lại, chuyển đổi tệp dữ liệu kiểu text thành một tệp dữ liệu theo khuôn dạng DBF.

7.12. Tìm hiểu cách sắp xếp và tìm kiếm theo kiểu index của các hệ quản trị cơ sở dữ liệu như foxpro hoặc access.

PHỤ LỤC

Cách học lập trình tốt nhất là viết chương trình. Đây là bài tập tổng kết lại toàn bộ những kiến thức cơ bản nhất đã đề cập trong giáo trình và được thể hiện lại bằng một chương trình. Chúng tôi hy vọng rằng, sinh viên sẽ hoàn thành nó trước khi thi hết học phần của giáo trình này.

Bài toán: Xây dựng hệ thống thực hiện những nhiệm vụ sau:

1. Xây dựng tập thao tác trên tập các ký tự:

Cho một byte ký tự bất kỳ. Hãy xây dựng những thao tác sau trên byte ký tự đó:

1.1. Nhận một byte ký tự;

1.2. Đổi ký tự in thường thành ký tự in hoa;

1.3. Đổi ký tự in hoa thành ký tự in thường;

1.4. In ra mã nhị phân của ký tự;

1.5. Chẵn hoá byte lẻ theo kỹ thuật parity;

1.6. Giải mã byte chẵn theo kỹ thuật parity;

2. Xây dựng tập thao tác trên tập các số nguyên dương a, b:

Cho hai số nguyên dương a, b bất kỳ có biểu diễn nhị phân tương ứng là $a = (a_1, a_2, ..., a_n)$; $b = (b_1, b_2, ..., b_n)$. Hãy xây dựng tập thao tác sau trên hai số nguyên a và b.

2.1. Nhận hai số nguyên dương a, b;

2.2. Chuyển a, b thành số ở hệ cơ số bất kỳ;

2.3. Cộng hai số nguyên a, b;

2.4. Trừ hai số nguyên a, b;

2.5. Nhân hai số nguyên a, b;

2.6. Lấy phần dư a mod b;

2.7. Tìm ước số chung lớn nhất của a, b.

3. Xây dựng tập thao tác trên các xâu ký tự:

Cho hai xâu ký tự str1, str2. Trong đó, str1[i] ≠ str1[j], str2[i] ≠ str2[j] với mọi i ≠ j. Hãy xây dựng tập các thao tác sau trên str1 và str2.

3.1. Nhập hai xâu ký tự str1, str2;

3.2. Tìm hiệu hai xâu str1\str2;

3.3. Tìm tích hợp hai xâu str1∪str2;

3.4. Tìm giao hai xâu str1∩str2;

3.5. Sắp xếp str1∪str2 theo thứ tự từ điển các ký tự;

3.6. Tìm ký tự x bất kỳ có thuộc str1∪str2 hay không bằng kỹ thuật binary;

4. Xây dựng tập thao tác trên mảng số (đa thức):

Cho đa thức A bậc n, đa thức B bậc m với các hệ số thực. Hãy xây dựng tập thao tác sau cho hai đa thức:

4.1. Tạo lập đa thức A bậc n, đa thức B bậc n;

4.2. Tính giá trị đa thức A, B tại x_0 cho trước;

4.3. Tìm đạo hàm cấp l của A, B;

4.4. Tìm đa thức tổng C = A + B;

4.5. Tìm đa thức hiệu C = A- B;

4.6. Tìm đa thức tích C = A*B;

4.7. Tìm đa thức thương và đa thức phần dư của A/B.

5. Xây dựng tập thao tác trên mảng các xâu ký tự:

Cho mảng các xâu ký tự được lưu trong một buffer. Trong đó, mỗi xâu ký tự là một từ khoá của ngôn ngữ lập trình C. Hãy xây dựng các thao tác sau trên tập các từ khoá:

5.1. Tạo lập mảng các từ khoá;

5.2. Tìm từ khoá dài nhất;

5.3. Tìm từ khoá ngắn nhất;

5.4. Sắp xếp mảng từ khoá theo thứ tự từ điển;

5.5. Tìm kiếm một từ khoá theo kỹ thuật Binary;

5.6. Mã hoá mảng từ khoá theo kỹ thuật parity;

5.7. Giải mã mảng từ khoá theo kỹ thuật parity.

6. Xây dựng tập thao tác trên ma trận vuông:

Cho hai ma trận vuông A, B cấp n. Hãy xây dựng tập thao tác sau trên hai ma trận vuông A và B.

6.1. Tạo lập hai ma trận vuông A, B cấp n;

6.2. Tìm hàng, cột, đường chéo có tổng số các phần tử là lớn nhất;

6.3. Tìm ma trận tổng C = A + B;

6.4. Tìm ma trận tích C = A * B;

6.5. Tìm ma trận chuyển vị A^T, B^T;

6.6. Tìm ma trận nghịch đảo của A, B.

7. Xây dựng tập thao tác trên cấu trúc:

Cho hai số phức a, b. Mỗi số phức được định nghĩa như một cấu trúc gồm hai thành phần (phần thực và phần ảo). Hãy xây dựng tập thao tác sau trên cấu trúc.

7.1. Tạo lập hai số phức a, b;

7.2. Tìm tổng hai số phức c = a + b;

7.3. Tìm hiệu hai số phức c = a- b;

7.4. Tìm tích hai số phức c = a * b;

7.5. Tìm thương hai số phức c = a/b;

7.6. Tìm căn bậc hai của số phức a, b.

8. Xây dựng tập thao tác trên file:

Cho tệp văn bản được tổ chức thành từng dòng. Hãy xây dựng tập thao tác sau trên tệp văn bản đó.

8.1. Đếm số dòng trong tệp văn bản;

8.2. Đếm số từ trong tệp văn bản;

8.3. Tìm dòng dài nhất trong tệp;

8.4. Tìm từ dài nhất trong tệp;

8.5. Tìm tần số xuất hiện của từ x bất kỳ trong tệp;

8.6. Mã hoá tệp theo kỹ thuật caresa;

8.7. Giải mã tệp theo kỹ thuật caresa.

9. Xây dựng tập thao tác trên stack:

9.1. Tạo lập một stack;

9.2. Kiểm tra tính rỗng của stack;

9.3. Kiểm tra tính đầy của stack;

9.4. Thêm phần tử vào stack;

9.5. Loại bỏ phần tử trong stack;

9.6. Duyệt stack.

10. Xây dựng tập thao tác trên hàng đợi:

10.1. Tạo lập một hàng đợi;

10.2. Kiểm tra tính rỗng của hàng đợi;

10.3. Kiểm tra tính đầy của hàng đợi;

10.4. Thêm phần tử vào hàng đợi;

10.5. Loại bỏ phần tử trong hàng đợi;

10.6. Duyệt hàng đợi.

11. Xây dựng tập thao tác trên tập danh sách móc nối đơn:

Hãy xây dựng chương trình quản lý điểm của sinh viên bằng một danh sách móc nối đơn. Bao gồm:

11.1. Tạo lập danh sách móc nối đơn;

11.2. Thêm một phần tử vào đầu danh sách móc nối đơn;

11.3. Thêm một phần tử vào cuối danh sách móc nối đơn;

11.4. Thêm một phần tử vào giữa danh sách móc nối đơn;

11.5. Loại bỏ phần tử ở đầu danh sách móc nối đơn;

11.6. Loại bỏ phần tử ở cuối danh sách móc nối đơn;

11.7. Loại bỏ phần tử ở cuối danh sách móc nối đơn;

11.8. Duyệt danh sách móc nối đơn;

11.9. Sắp xếp theo các phần tử của danh sách móc nối đơn theo một khoá náo đó;

11.10. Tìm kiếm theo các phần tử của danh sách theo một khoá nào đó;

11.11. In bảng xếp loại sinh viên.

12. Xây dựng tập thao tác trên danh sách móc nối kép:

12.1. Tạo lập danh sách móc nối kép;

12.2. Thêm một phần tử vào đầu danh sách móc nối kép;

12.3. Thêm một phần tử vào cuối danh sách móc nối kép;

12.4. Thêm một phần tử vào giữa danh sách móc nối kép;

12.5. Loại bỏ phần tử ở đầu danh sách móc nối kép;

12.6. Loại bỏ phần tử ở cuối danh sách móc nối kép;

12.7. Loại bỏ phần tử ở cuối danh sách móc nối kép;

12.8. Duyệt danh sách móc nối kép;

12.9. Sắp xếp theo các phần tử của danh sách móc nối kép theo một khoá náo đó;

12.10. Tìm kiếm theo các phần tử của danh sách theo một khoá nào đó;

13. Xây dựng tập thao tác trên cây nhị phân:

13.1. Tạo lập một cây nhị phân;

13.2. Thêm một node vào nhánh bên trái của một lá;

13.3. Thêm một node vào nhánh bên phải của một lá;

13.4. Loại bỏ node lá nhánh bên trái của một node;

13.5. Loại bỏ node lá nhánh bên phải của một node;

13.6. Loại bỏ cả cây;

13.7. Xác định độ cao của cây;

13.8. Tìm kiếm node trên cây;

13.9. Duyệt cây theo thứ tự trước;

13.10. Duyệt cây theo thứ tự sau;

13.11. Duyệt cây theo thứ tự giữa;

14. Xây dựng tập thao tác trên cây nhị phân tìm kiếm:

14.1. Tạo lập node gốc của cây nhị phân tìm kiếm;

14.2. Thêm một node vào cây nhị phân tìm kiếm;

14.3. Loại bỏ node trái trái của một node;

14.4. Loại bỏ node bên phải của một node;

14.5. Loại bỏ cả cây;

14.6. Xác định độ cao của cây;

14.7. Tìm kiếm node trên cây;

14.8. Duyệt cây theo thứ tự trước;

14.9. Duyệt cây theo thứ tự sau;

14.10. Duyệt cây theo thứ tự giữa;

15. Duyệt và đệ qui:

15.1. Duyệt các tập con của tập gồm n phần tử;

15.2. Duyệt mọi tập con m phần tử từ tập gồm n phần tử;

15.3. Duyệt mọi hoán vị của tập gồm n phần tử;

15.4. Duyệt mọi cách phân chia số tự nhiên n thành tổng các số tự nhiên nhỏ hơn;

15.5. Duyệt mọi bộ giá trị trong tập các giá trị rời rạc;

15.6. Giải bài toán cái túi;

15.7. Giải bài toán người du lịch.

16. Đường đi chu trình trên đồ thị:

Cho đồ thị G = <V, E> trong đó V là tập đỉnh, E là tập cạnh được biểu diễn như một ma trận kề A_{ij}. Hãy xây dựng tập thao tác sau trên đồ thị:

16.1. Duyệt đồ thị theo BFS;

16.2. Duyệt đồ thị theo DFS;

16.3. Tìm số thành phần liên thông của đồ thị theo DFS;

16.4. Tìm số thành phần liên thông của đồ thị theo BFS;

16.5. Tìm đường đi từ đỉnh i đến đỉnh j của đồ thị theo DFS;

16.6. Tìm đường đi từ đỉnh i đến đỉnh j của đồ thị theo BFS;

17. Đồ thị Euler & đồ thị Hamilton:

Cho đồ thị G = <V, E> trong đó V là tập đỉnh, E là tập cạnh được biểu diễn như một ma trận kề A_{ij}. Hãy xây dựng tập thao tác sau trên đồ thị:

17.1. Kiểm tra tính liên thông của đồ thị;

17.2. Kiểm tra đồ thị có chu trình Euler hay không;

17.3. Tìm một chu trình Euler của đồ thị;

17.4. Liệt kê mọi chu trình Euler của đồ thị;

17.5. Kiểm tra đồ thị có đường đi Euler hay không;

17.6. Liệt kê mọi đường đi Euler của đồ thị;

17.7. Liệt kê mọi chu trình Hamilton của đồ thị;

17.8. Liệt kê mọi đường đi Hamilton của đồ thị.

18. Đồ thị trọng số:

18.1. Tìm một cây bao trùm của đồ thị vô hướng liên thông theo thuật toán BFS;

18.2. Tìm một cây bao trùm của đồ thị vô hướng liên thông theo thuật toán DFS;

18.3. Tìm cây bao trùm nhỏ nhất trên đồ thị vô hướng liên thông có trọng số (thuật toán Kruskal);

18.4. Tìm đường đi ngắn nhất xuất phát tại một đỉnh (thuật toán Ford-Bellman);

18.5. Tìm đường đi ngắn nhất xuất phát từ một đỉnh tới tất cả các đỉnh khác của đồ thị trong trường hợp đồ thị có trọng số không âm (Thuật toán Dijkstra);

18.6. Tìm đường đi ngắn nhất giữa tất cả các cặp đỉnh (thuật toán Floyd).

19. Các giải pháp sắp xếp và tìm kiếm:

19.1. Sắp xếp kiểu lựa chọn;

19.2. Sắp xếp kiểu sủi bọt;

19.3. Sắp xếp kiểu trèn trực tiếp;

19.4. Sắp xếp nhanh Quick Sort;

19.5. Sắp xếp kiểu vun đồng Heap Sort;

19.6. Tìm kiếm tuyến tính;

19.7. Tìm kiếm nhị phân;

19.8. Tìm kiếm trên cây.

TÀI LIỆU THAM KHẢO

[1] N. Knuth. *Computer Programming*, McGraw Hill, 1996.

[2] N. Knuth. *Algorithms*, McGraw Hill, 1995.

[3] Reingold E.M, Nievergetlt J, Deo N. *Combinatorial Algorithms. Theory and practise*. Prentice Hall, 1997.

[4] Brian Berninghan, Denit Ritche. *Discrete Mathematics*. Prentice Hall, 1997.

[5] Harary F. *Graph Theory*. Addison Wesley Publishing Ecadamic Press, 1983.

[6] Brian Kerninghan, Denist Ritche. *C Language. Norm ANSI*. Prentice Hall, 1988.

[7] Hall M. *Combinatorial Theory*. Blaisdell Publishing Company. London, 1967.

[8] Bryon Gottfried. *Programming With C*. McGraw Hill, 1996.

[9] Carl Townsend. *Understanding C*. SAMS 1989.

[10] Paul Davies. *The Inspensable Guide to C*. Addision Wisley, 1996.

[11] Nikolus L.R. Wirth. *Program = Data Structure + Algorithms*. Prentice Hall, 1992.

[12] Đặng Huy Ruận. *Lý thuyết đồ thị*. Nhà xuất bản khoa học kỹ thuật Hà Nội, 2000.

[13] Đỗ Xuân Lôi. *Cấu trúc dữ liệu và giải thuật*. Nhà xuất bản Khoa học kỹ thuật Hà Nội, 1994

[14] Ngô Trung Việt. *Ngôn ngữ lập trình C*, Nhà xuất bản Khoa học kỹ thuật Hà Nội, 1990.

[15] Nguyễn Hồng Chương. *Cấu trúc dữ liệu ứng dụng và cài đặt bằng C*. Nhà xuất bản Thành phố Hồ Chí Minh, 1999.

[16] Nguyễn Đức Nghĩa, Nguyễn Bình Thành. Toán học rời rạc. Nhà xuất bản Khoa học kỹ thuật Hà nội, 1998.

Giáo trình

KỸ THUẬT LẬP TRÌNH

Chịu trách nhiệm xuất bản

LƯU ĐỨC VĂN

Biên tập : TRẦN VŨ THƯỞNG - ĐẶNG HUY TOÀN

Chế bản : TRẦN VŨ THƯỞNG - BÙI NGỌC KHOA

Sửa bản in : ĐẶNG HUY TOÀN

Trình bày bìa: BÙI NGỌC KHOA

(Giáo trình này được ban hành kèm theo Quyết định số 479/QĐ-QLNCKH ngày 26 tháng 9 năm 2002 của Giám đốc Học viện Công nghệ Bưu chính Viễn thông)

NHÀ XUẤT BẢN BƯU ĐIỆN

Trụ sở: 18 - Nguyễn Du, Hà Nội
Điện thoại: 04-9431283, 9432438 Fax: 04-9431285
E-mail: nxbbuudien@mpt.gov.vn Website: www.nxbbuudien.com.vn

Chi nhánh TP. HCM: 27 - Nguyễn Bỉnh Khiêm, Quận 1, TP. Hồ Chí Minh
Điện thoại: 08-9100925 Fax: 08-9100924
E-mail: chinhanh-nxbbd@hcm.vnn.vn

Chi nhánh TP. Đà Nẵng: 42 - Trần Quốc Toản, Quận Hải Châu, TP. Đà Nẵng
Điện thoại: 0511-897467 Fax: 0511-897467
E-mail: pnbich@mpt.gov.vn

28(I) – 283

In 500 bản, khổ 16x24 cm, tại Công ty In Khoa học Kỹ thuật
Số đăng ký kế hoạch xuất bản: 96-2006/CXB/59_4-08/BuĐ
Số quyết định xuất bản:185/QĐ-NXB BĐ
In xong và nộp lưu chiểu tháng 11 năm 2006.

NHÀ XUẤT BẢN BƯU ĐIỆN

Trụ sở: 18 - Nguyễn Du, Hà Nội
Điện thoại: 04-9432438, 9431284 Fax: 04-9431285
E-mail: bientap@hn.vnn.vn Website: www.nxbbuudien.com.vn

Chi nhánh TP. HCM: 27 - Nguyễn Bỉnh Khiêm, Quận 1, TP.Hồ Chí Minh
Điện thoại: 08-9100925 Fax: 08-9100924
E-mail: chinhanh-nxbbd@hcm.vnn.vn

Chi nhánh TP. Đà Nẵng: 42 - Trần Quốc Toản, Quận Hải Châu, TP. Đà Nẵng
Điện thoại: 0511-897497 Fax: 0511-897497
E-mail: pnbich@mpt.gov.vn

Giá: 57.000đ