

BÀI GIẢNG
KỸ THUẬT LẬP TRÌNH

Biên soạn : Ths. NGUYỄN DUY PHƯƠNG

GIỚI THIỆU MÔN HỌC

I. GIỚI THIỆU CHUNG

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 đầy tiềm năng. 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, tiến tới phát triển cũng như làm chủ các hệ thống phần mềm của các kỹ sư điện tử viễn thông là rất cần thiết.

Tài liệu giảng dạy “*Kỹ thuật lập trình*” cho hệ đào tạo từ xa được xây dựng dựa trên giáo trình “*Kỹ thuật lập trình*” đã được giảng dạy tại học viện trong những năm qua với mục đích cung cấp cho sinh viên những kiến thức cơ bản nhất, có tính hệ thống liên quan tới lập trình.

Thông qua cuốn tài liệu này, chúng tôi muốn giới thiệu với các bạn đọc về kỹ năng lập trình cấu trúc và một số thuật toán quan trọng, bao gồm: Đại cương về lập trình cấu trúc; 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.

II. MỤC ĐÍCH

Môn học cung cấp cho sinh viên kỹ năng lập trình trên các cấu trúc dữ liệu quan trọng như: stack, queue, link, tree & graph cùng với phương pháp phân tích, thiết kế, đánh giá thuật toán.

Sau khi học xong môn học này, sinh viên có khả năng viết được chương trình giải quyết những bài toán trong thực tế.

III. PHẠM VI NGHIÊN CỨU

Nghiên cứu các thuật toán cơ bản được sử dụng trong thực tế như các thuật toán tìm kiếm, các thuật toán liên quan đến đồ thị. Các giải thuật lập trình dựa trên danh sách, cây...

Nghiên cứu cách cài đặt các thuật toán trên máy tính.

Tìm hiểu các lĩnh vực ứng dụng của các thuật toán, phương pháp trong thực tế.

IV. PHƯƠNG PHÁP NGHIÊN CỨU

Để học tốt môn học này, sinh viên cần lưu ý những vấn đề sau:

1. Kiến thức cần trước

- Sinh viên phải có kiến thức cơ bản về toán học cao cấp.
- Thành thạo ít nhất một ngôn ngữ lập trình. Đặc biệt trong cuốn sách này đã sử dụng ngôn ngữ lập trình C để mô tả thuật toán, vì vậy sinh viên phải nắm được ngôn ngữ lập trình C.

2. Các tài liệu cần có:

Sách hướng dẫn học tập Kỹ thuật lập trình. Ths. Nguyễn Duy Phương, Học viện Công nghệ Bưu chính Viễn thông, 2006.

Nếu cần sinh viên nên tham khảo thêm:

- Giáo trình Kỹ thuật lập trình. Ts. Lê Hữu Lập, Ths. Nguyễn Duy Phương, Học viện Công nghệ Bưu chính Viễn thông, 2002.
- Bài giảng điện tử môn học: “Kỹ thuật lập trình” của Học viện Công nghệ Bưu chính Viễn thông.

3. Đặt ra mục tiêu, thời hạn cho bản thân

Đặt ra các mục tiêu tạm thời và thời hạn cho bản thân và cố gắng thực hiện chúng

Xây dựng mục tiêu trong chương trình nghiên cứu.

4 Nghiên cứu và nắm những kiến thức cốt lõi

Sinh viên nên đọc qua sách hướng dẫn học tập trước khi nghiên cứu bài giảng môn học và các tài liệu tham khảo khác.

5. Tham gia đầy đủ các buổi hướng dẫn học tập

Thông qua các buổi hướng dẫn học tập, giảng viên sẽ giúp sinh viên nắm được nội dung tổng thể của môn học và giải đáp thắc mắc, đồng thời sinh viên cũng có thể trao đổi, thảo luận với những sinh viên khác về nội dung bài học.

6. Chủ động liên hệ với bạn học và giảng viên

Cách đơn giản nhất là tham dự các diễn đàn học tập trên mạng Internet, qua đó có thể trao đổi trực tiếp các vấn đề vướng mắc với giảng viên hoặc các bạn học khác đang online.

7. Tự ghi chép lại những ý chính

Việc ghi chép lại những ý chính là một hoạt động tái hiện kiến thức, kinh nghiệm cho thấy nó giúp ích rất nhiều cho việc hình thành thói quen tự học và tư duy nghiên cứu.

8. Học đi đôi với hành

Học lý thuyết đến đâu thực hành làm bài tập và **thực hành** ngay đến đó để hiểu và nắm chắc lý thuyết. Sinh viên cần cài đặt trên máy tính các thuật toán trong bài học bằng các ngôn ngữ lập trình để từ đó có thể hiểu và nắm chắc hơn tư tưởng và nội dung của thuật toán.

Hà Nội, ngày 20 tháng 02 năm 2006

Ths. Nguyễn Duy Phương

CHƯƠNG 1: ĐẠI CƯƠNG VỀ KỸ THUẬT LẬP TRÌNH CẤU TRÚC

Nội dung chính của chương này tập chung làm sáng tỏ những nguyên lý cơ bản của lập trình cấu trúc. Những nguyên lý này được coi như nền tảng tư tưởng của phương pháp lập trình cấu trúc đã được tích hợp trong các ngôn ngữ lập trình. Nắm vững các nguyên lý của lập trình cấu trúc không chỉ giúp người học có cách tiếp cận ngôn ngữ lập trình nhanh chóng mà còn giúp họ cách tư duy trong khi xây dựng các hệ thống ứng dụng. Các nguyên lý cơ bản được giới thiệu trong chương này bao gồm:

- ✓ Nguyên lý lệnh - lệnh có cấu trúc - cấu trúc dữ liệu.
- ✓ Nguyên lý tối thiểu.
- ✓ Nguyên lý địa phương.
- ✓ Nguyên lý an toàn.
- ✓ Nguyên lý nhất quán.
- ✓ Nguyên lý Top-Down .
- ✓ Nguyên lý Botton-Up.

Bạn đọc có thể tìm được những chi tiết sâu hơn và rộng hơn trong tài liệu [1] & [6].

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. 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 màn hình che mang tính chất thương mại được gọi là Version. 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 trúc vật lý cụ thể của hệ thống máy tính, tạo nên một số sản phẩm 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 trúc 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:

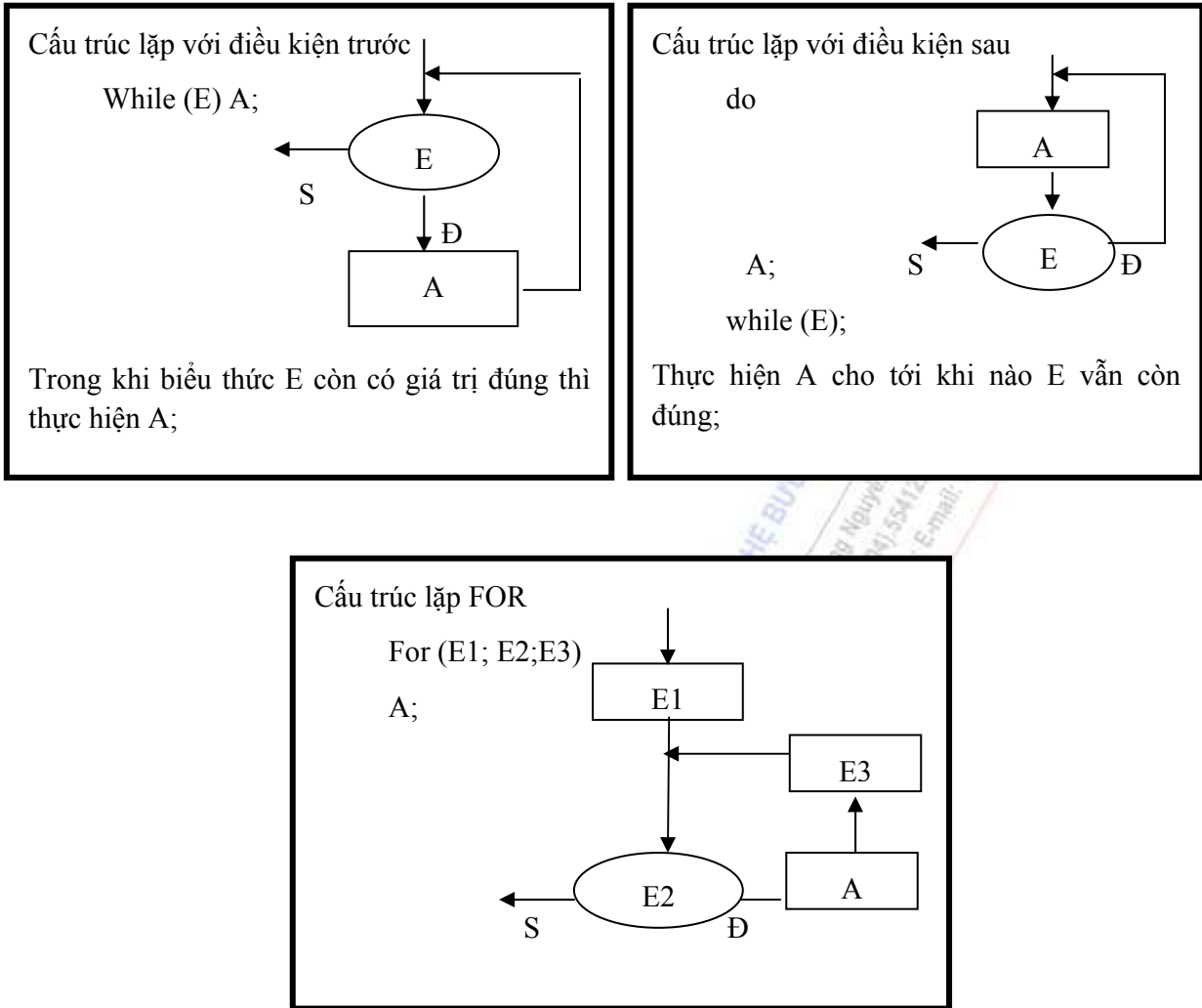
$DO\ 50\ I = 12.523$

(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 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 hủy 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 cùng với lập trình viên là đối tượng nghiên cứu của kỹ thuật lập trình và phương pháp



Hình 1.2. Các cấu trúc lặp

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, tuyến 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 kí 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 chuỗi 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 một đơn vị;

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

$a-=n \Leftrightarrow a = a - n;$ // giảm giá trị biến nguyên a n đơn vị);

$a\%=n \Leftrightarrow a = a\%n;$ // lấy giá trị biến a modul với n;

$a/=n \Leftrightarrow a=a/n;$ // lấy giá trị biến a chia cho n;

$a*=n \Leftrightarrow a = a*n;$ // 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:

$\text{if} (a>b) \{ \dots \}$ // nếu a lớn hơn b

$\text{if} (a<b) \{ \dots \}$ // nếu a nhỏ hơn b

$\text{if} (a>=b) \{ \dots \}$ // nếu a lớn hơn hoặc bằng b

$\text{if} (a<=b) \{ \dots \}$ // nếu a nhỏ hơn hoặc bằng b

$\text{if} (a==b) \{ \dots \}$ // nếu a đúng bằng b

$\text{if} (a!=b) \{ \dots \}$ // 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 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:

$\text{int } a = 3, b = 5;$

$\text{if} ((a != 0) \&\& (b != 0))$ // nếu a khác 0 và b khác 0

$\text{if} ((a != 0) \| (b != 0))$ // nếu a khác 0 hoặc b khác 0

$\text{if} (! a)$ // phủ định a khác 0

$\text{if} (a == b)$ // 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.

\wedge : Phép tuyển các bit có loại trừ.

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

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

\sim : 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. Quy 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(voi)(
    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 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

1.3.2. Tập các lệnh vào ra cơ bản

Nhập dữ liệu từ bàn phím: `scanf("format_string, ...", ¶meter ...);`
 Nhập dữ liệu từ tệp: `fscanf(file_pointer, "format_string, ...", ¶meter, ...);`
 Nhận một ký tự từ bàn phím: `getch(); getchar();`
 Nhận một ký tự từ file: `fgetc(file_pointer, character_name);`
 Nhập một string từ bàn phím: `gets(string_name);`
 Nhận một string từ file text : `fgets(string_name, number_character, file_pointer);`
 Xuất dữ liệu ra màn hình: `printf("format_string ...", parameter ...);`
 Xuất dữ liệu ra file : `fprintf(file_pointer, "format_string ...", parameter. . .);`
 Xuất một ký tự ra màn hình: `putch(character_name);`
 Xuất một ký tự ra file: `fputc(file_pointer, character_name);`
 Xuất một string ra màn hình: `puts(const_string_name);`
 Xuất một string ra file: `fputs(file_pointer, const_string_name);`

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

Tập thao tác trên string:

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

*int strcmp(const char *s1, const char *s2)* : so sánh hai chuỗi 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 chuỗi src vào sau chuỗi dest.

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

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

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

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

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

Tập thao tác trên con trỏ:

Thao tác lấy địa chỉ của biến: *¶meter_name*;

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

Thao tác trở tới phần tử tiếp theo: *++pointer_name*;

Thao tác trở tới phần tử thứ n kể từ vị trí hiện tại: *pointer_name = pointer_name + n*;

Thao tác trở tới phần tử sau con trỏ kể từ vị trí hiện tại: *--pointer_name*;

Thao tác trở 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)*;

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_name1 = struct_parameter_name2;

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

pointer_struct_parameter_name -> struct_parameter_name.

Tập thao tác trên file:

Khai báo con trỏ file: *FILE *file_pointer;*

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

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

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

Thao tác đọc từng khối trong file:

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

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

Thao tác ghi từng khối vào file:

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

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

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

Thao tác loại bỏ file: *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 khi 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 cục.
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 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.

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 phẩy 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 file *.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: 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 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 compiler 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 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 1.6. 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 Da thuc %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 Nhap n="); scanf("%d", n);*n=*n+1;
    printf("\n Nhap m="); scanf("%d",m); *m=*m+1;
    printf("\n Nhap he so da thuc A:");
    for(i=0; i<*n;i++){
        printf("\n A[%d]=", i); scanf("%f", &temp);
        A[i]=temp;
    }
    printf("\n Nhap he so da thuc 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 BAC 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

- *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 $a=(a_1, a_2, \dots, a_n)$, $b = (b_1, b_2, \dots, b_n)$; $a_i, b_i = 0, 1, i=1, 2, \dots, n$. Hãy xây dựng tập thao tác trên hai số nguyên đó.

Mức tổng quan (level 0):

Hình dung toàn bộ những thao tác trên hai số nguyên $a=(a_1, a_2, \dots, a_n)$, $b=(b_1, b_2, \dots, b_n)$ 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 : $a=(a_1, a_2, \dots, a_n)_b$; (*khai triển cơ số b bất kỳ*)
Format : Binary(a);
Actions
{ $Q = n; k=0;$
 While ($Q \neq 0$) {
 $a_k = q \text{ mod } b;$
 $q = q \text{ div } b;$
 $k = k + 1;$
 }
 < Khai triển cơ số b của a là $(a_{k-1}, a_{k-2}, \dots, a_1, a_0)$ >;
}

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

Input:
 $a=(a_1, a_2, \dots, a_n)$,
 $b = (b_1, b_2, \dots, b_n)$;
Output:

```
c = a + b;  
Format: Addition(a, b);  
Actions {  
    c = 0;  
    for (j = 0; j < n; j++) {  
  
        d = (aj + bj + c) div 2;  
        sj = aj + bj + c - 2d;  
        c = d;  
    }  
    sn = c;  
    < Khai triển nhị phân của tổng là (snsn-1 . . . s1s0)2 >  
}
```

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

```
Input:  
    a=(a1, a2, . . . , an),  
    b = (b1, b2, . . . , bn);  
Output:  
    c = a - b;  
Format: Subtraction(a, b);  
Actions {  
    b = -b;  
    c = Addition(a, b);  
    return(c);  
}
```

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

```
Input:  
    a=(a1, a2, . . . , an),  
    b = (b1, b2, . . . , bn);  
Output:  
    c = a * b;  
Format: Mutual(a, b);  
Actions {  
    For (j =0; j < n; j++) {  
        If ( bj =1)  
            cj = a<<j;  
        Else  
            cj = 0;  
    }  
    (* c0, c1, . . . , cn-1 là các tích riêng*)  
    p=0;
```

```
for( j=0; j< n; j++) {  
    p = Addition(p, cj);  
}  
return(p);  
}
```

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

Input:

a=(a₁, a₂, . . . , a_n),

b = (b₁, b₂, . . . , b_n);

Output:

c = a div b;

Format: Division(a, b);

Actions {

c = 0;

while (a >= b) {

c = c + 1;

a = Subtraction(a, b);

}

return(c);

}

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

Input:

a=(a₁, a₂, . . . , a_n),

b = (b₁, b₂, . . . , b_n);

Output:

c = a mod b;

Format: Modulation(a, b);

Actions {

while (a >= b)

a = Subtraction(a, b);

return(a);

}

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

Input:

a=(a₁, a₂, . . . , a_n),

b = (b₁, b₂, . . . , b_n);

Output:

c = USCLN(a,b);

Format: USCLN(a, b);

Actions {

```
while ( a≠ b ) {  
    if ( a > b )  
        a = Subtraction(a, b)  
    else  
        b = Subtraction(b,a);  
}  
return(a);  
}
```

Để ý rằng, sau khi phân rã bài toán ở mức 1, chúng ta chỉ cần xây dựng hai phép toán cộng và phép tính nhân các số nhị phân của a, 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 kế 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 Botton-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 Nhap a=");scanf("%d", a);  
    printf("\n Nhap 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);
}
```

TRƯỜNG ĐẠI HỌC CÔNG NGHỆ BƯU CHÍNH VIỄN THÔNG
Km10 Đường Nguyễn Trãi, Hà Đông-Hà Tây
Tel: (04) 5541221; Fax: (04) 5540587
Website: <http://www.c-pit.edu.vn>; E-mail: dhkc@pit.edu.vn

TRƯỜNG ĐẠI HỌC PTIT
ĐÀO TẠO ĐẠI HỌC TỪ XA

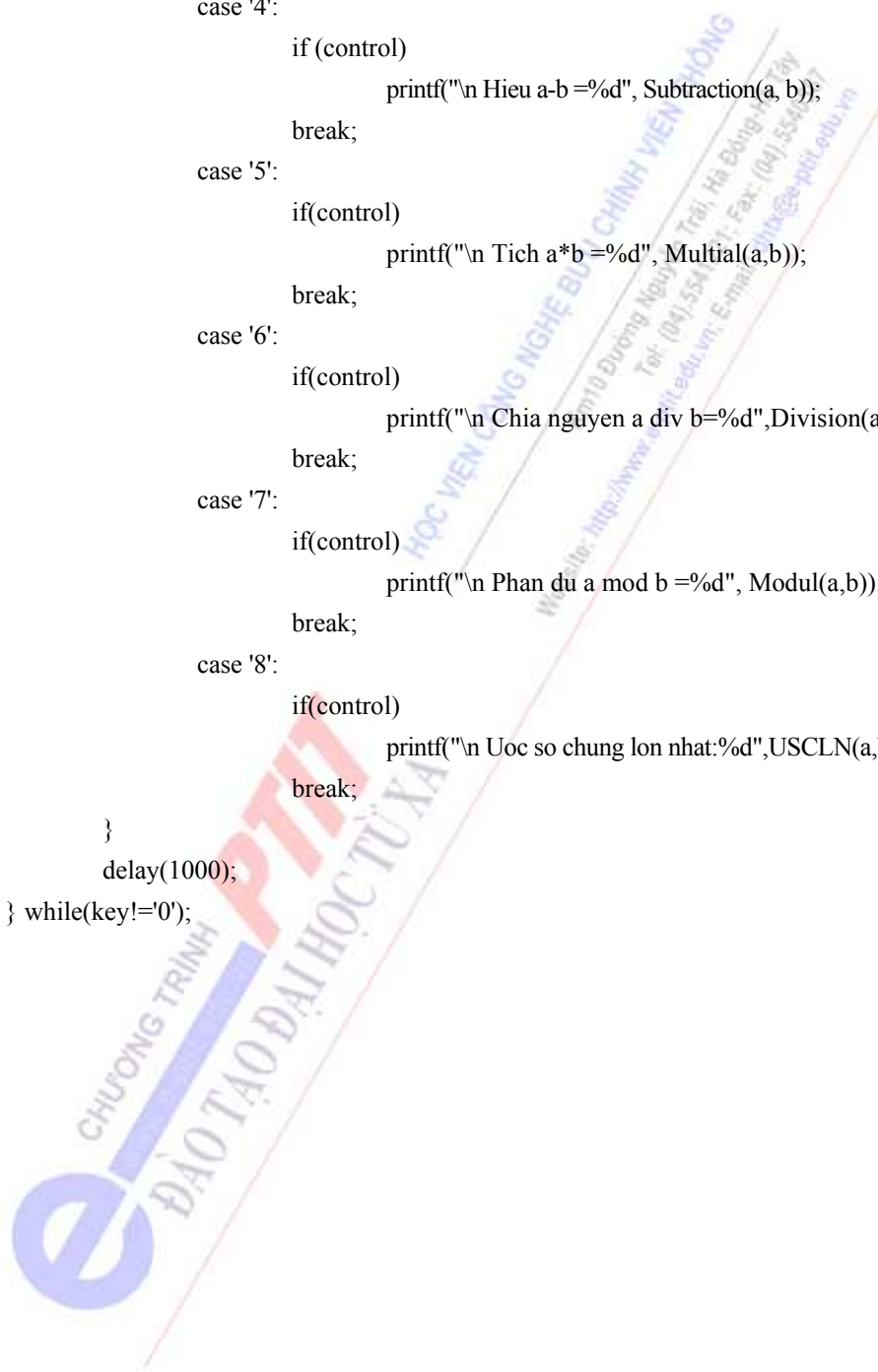

```
}
int Modul(int a, int b){
    while(a>=b)
        a = Subtraction(a,b);
    return(a);
}
int Division(int a, int b){
    int d=0;
    while(a>=b) {
        a= Subtraction(a,b);
        d++;
    }
    return(d);
}
int USCLN(int a, int b){
    while(a!=b){
        if(a>b) a = Subtraction(a,b);
        else b = Subtraction(b,a);
    }
    return(a);
}
void main(void){
    int a, b, key, control=0;
    do {
        clrscr();
        printf("\n Tap thao tac voi so nguyen");
        printf("\n 1- Nhap hai so a,b");
        printf("\n 2- So nhi phan cua a, b");
        printf("\n 3- Tong hai so a,b");
        printf("\n 4- Hieu hai so a,b");
        printf("\n 5- Tich hai so a,b");
        printf("\n 6- Thuong hai so a,b");
        printf("\n 7- Phan du hai so a,b");
        printf("\n 8- USCLN hai so a,b");
        printf("\n 0- Tro ve");
        key=getch();
        switch(key){
            case '1': Init(&a, &b); control=1; break;
            case '2':
                if (control){
                    Binary(a); Binary(b);
                }
        }
    }
}
```

HỌC VIỆN CÔNG NGHỆ BƯU CHÍNH VIỄN THÔNG

Km10 Đường Nguyễn Trãi, Hà Đông-Hà Tây
Tel: (04) 5541221; Fax: (04) 5540587

Website: <http://www.e-ptit.edu.vn>; E-mail: dhk@ptit.edu.vn

```
        }
        break;
    case '3':
        if (control)
            printf("\n Tong a+b = %d", Addition(a, b));
        break;
    case '4':
        if (control)
            printf("\n Hieu a-b =%d", Subtraction(a, b));
        break;
    case '5':
        if(control)
            printf("\n Tich a*b =%d", Multial(a,b));
        break;
    case '6':
        if(control)
            printf("\n Chia nguyen a div b=%d",Division(a,b));
        break;
    case '7':
        if(control)
            printf("\n Phan du a mod b =%d", Modul(a,b));
        break;
    case '8':
        if(control)
            printf("\n Uoc so chung lon nhat:%d",USCLN(a,b));
        break;
    }
    delay(1000);
} while(key!='0');
}
```



NHỮNG NỘI DUNG CẦN GHI NHỚ

- ✓ Một ngôn ngữ lập trình bất kỳ đều dựa trên tập các cấu trúc lệnh điều khiển (tuần tự, tuyển chọn & lặp), các cấu trúc dữ liệu (dữ liệu kiểu cơ bản & dữ liệu có cấu trúc) cùng với các phép toán trên nó.
- ✓ Khi mới bắt đầu học lập trình, hãy lập trình từ tập tối thiểu các công cụ mà ngôn ngữ lập trình trang bị (Nguyên lý tối thiểu).
- ✓ Khi nào dùng biến địa phương, khi nào dùng biến toàn cục là nội dung chính của nguyên lý địa phương. Nắm vững nguyên lý này giúp cho ta sử dụng cách truyền tham biến và cách truyền tham trị cho hàm.
- ✓ Dữ liệu kiểu nào thì phép toán đó là nội dung chính của nguyên lý nhất quán.
- ✓ Mọi lỗi dù nhỏ nhất cũng phải lường trước ở mỗi mức cài đặt của chương trình.
- ✓ Cách phân rã một vấn đề lớn thành những vấn đề nhỏ hơn là nội dung chính của nguyên lý Top-Down.
- ✓ Cách cài đặt một vấn đề được thực hiện từ mức đơn vị chương trình (hàm, thủ tục) đến mức lắp ghép các đơn vị chương trình thành hệ thống hoàn thiện là nội dung chính của nguyên lý Botton-Up.



BÀI TẬP CHƯƠNG 1

Bài 1. Tìm các nghiệm nguyên dương của hệ phương trình:

$$X + Y + Z = 100$$

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

Bài 2. 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 $a \leq b \leq c \leq n$.

Bài 3. Cho số tự nhiên n . Hãy tìm các số Fibonacci nhỏ hơn n . Trong đó các số Fibonacci được định nghĩa như sau:

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

Bài 4. Chứng minh rằng, với mọi số nguyên dương $N, 0 < N \leq 39$ thì $N^2 + N + 41$ là một số nguyên tố. Điều khẳng định trên không còn đúng với $N > 39$.

Bài 5. Cho số tự nhiên n . Hãy liệt kê tất cả các số nguyên tố nhỏ hơn n .

Bài 6. Cho số tự nhiên n . Hãy tìm tất cả các số nguyên tố nhỏ hơn n bằng phương pháp sàng Estheven.

Bài 7. 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).

Bài 8. 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 .

Bài 9. 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 $1 \leq S \leq 45$.

Bài 10. Một số được gọi là số Mersenne 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 Mersenne nhỏ hơn n .

Bài 11. Cho số tự nhiên n . Hãy phân tích n thành tích các thừa số nguyên tố. Ví dụ $12 = 2 \cdot 2 \cdot 3$.

Bài 12. 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 $[P, Q]$.

Bài 13. Cho số tự nhiên n . Hãy tìm tất cả các số $1, 2, \dots, n$ sao cho các số trùng với phần cuối bình phương chính nó (Ví dụ : $62 = 36, 252 = 625$).

Bài 14. 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.

Bài 15. 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.

Bài 16. 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.

Bài 17. 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 . .9, Y là các chữ số nhận giá trị hoặc 0 hoặc là 1, N là các chữ số nhận giá trị từ 2 . .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 File 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 vài dấu trống.

Bài 18. Cho File dữ liệu TEXT.TXT được tổ chức thành từng dòng, độ dài tối đa của mỗi dòng là 80 ký tự. Kỹ thuật mã hoá tuyến tính là phương pháp biến đổi mã của các ký tự từ [a . .z], [A . .Z] thành một ký 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 File dữ liệu TEXT.TXT. Quá trình mã hoá được ghi lại trong File MAHOA.TXT, quá trình giải mã ghi lại trong File GIAIMA.TXT.

Bài 19. Cho File dữ liệu TEXT.TXT được tổ chức thành từng dòng, độ dài tối đa của mỗi dòng là 80 ký tự. Kỹ thuật mã hoá chẵn lẻ là phương pháp biến đổi mã của các ký tự [a . .z], [A . .Z]. Trong đó, nếu ký tự có số các bit 1 là lẻ ta bổ xung thêm một bit có giá trị một vào bit số 7 của ký tự để ký 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 File dữ liệu TEXT.TXT. Quá trình mã hoá được ghi lại trong File MAHOA.TXT, quá trình giải mã ghi lại trong File GIAIMA.TXT.

CHƯƠNG 2: DUYỆT VÀ ĐỆ QUI

Duyệt toàn bộ là phương pháp phổ dụng nhất trong khi giải quyết một bài toán trên máy tính. Các kỹ thuật duyệt cũng rất phong phú đa dạng nếu như ta chúng ta lợi dụng được những mẹo mực không mang tính tổng quát hoá nhưng hạn chế được không gian tìm kiếm lời giải bài toán. Đệ qui được sử dụng nhiều trong các kỹ thuật duyệt. Sử dụng đệ qui thường cho ta một lời giải tương đối ngắn gọn, dễ hiểu nhưng ẩn chứa trong nó nhiều bí ẩn khó lường. Tuy nhiên, nó vẫn được coi là một mẫu hình để vét cạn tất cả các khả năng của bài toán. Các kỹ thuật đệ qui được đề cập ở đây bao gồm:

- ✓ Các định nghĩa bằng đệ qui, các cấu trúc dữ liệu định nghĩa bằng đệ qui & giải thuật đệ qui.
- ✓ Thuật toán sinh kế tiếp giải quyết bài toán duyệt.
- ✓ Thuật toán quay lui giải quyết bài toán duyệt.
- ✓ Thuật toán nhánh cận giải quyết bài toán duyệt.

Bạn đọc có thể tìm thấy nhiều hơn những ứng dụng và cài đặt cụ thể phương pháp duyệt trong tài liệu [1].

2.1. 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 2.1. Để tính tổng $S(n) = 1 + 2 + \dots + n$, 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 $S(n-1) = S(n-2) + n-1$.
-
- 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 2.2. Định nghĩa hàm bằng đệ qui

Hàm $f(n) = n!$

Dễ thấy $f(0) = 1$.

Vì $(n+1)! = 1 \cdot 2 \cdot 3 \dots n(n+1) = n! (n+1)$, nên ta có:

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

Ví dụ 2.3. 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 2.4. Cấu trúc tự trở được định nghĩa bằng đệ qui

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

2.2. 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 $(b \bmod a)$ và a vì $USCLN(b \bmod a, a) = USCLN(a, b)$. 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 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));
}

```

2.3. 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_Kế_Tiếp* 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_Kế_Tiếp* sẽ gán cho *stop* giá trị *true*, ngược lại cấu hình kế tiếp sẽ được sinh ra.

```

Procedure generate{
    <Xây dựng cấu hình ban đầu>;
    stop =false;
    while (! stop) {
        <Đưa ra cấu hình đang có >;
        Sinh_Kế_Tiếp;
    }
}

```


}
}

Dưới đây là một ví dụ điển hình minh họa cho thuật toán sinh kế tiếp.

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 $1, 2, \dots, n$. Bài toán được đặt ra là: Cho một tập hợp gồm n phần tử $X = \{ X_1, X_2, \dots, X_n \}$, 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 chuỗi nhị phân có độ dài n là $B = \{ B_1, B_2, \dots, B_n \}$ 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 chuỗi nhị phân có độ dài n . Số các chuỗi nhị phân có độ dài n là 2^n . Bây giờ ta đi xác định thứ tự các chuỗi nhị phân và phương pháp sinh kế tiếp.

Nếu xem các chuỗi nhị phân $b = \{ b_1, b_2, \dots, b_n \}$ 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 chuỗi nhị phân $b = \{ b_1, b_2, \dots, b_n \}$ có thứ tự trước chuỗi nhị phân $b' = \{ b'_1, b'_2, \dots, b'_n \}$ và kí hiệu là $b < b'$ nếu $p(b) < p(b')$. Ví dụ với $n=4$: chúng ta có $2^4 = 16$ chuỗi 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 $i = n, n-1, \dots, 1$ 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 ( bi == 1 ) {
        bi = 0;
        i = i-1;
    }
    bi = 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 Xau nhi phan thu %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 Nhap n=");scanf("%d",&n);
    B =(int *) malloc(n*sizeof(int));
    Init(B,n);Generate(B,n);free(B);getch();
}

```

2.4. 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.

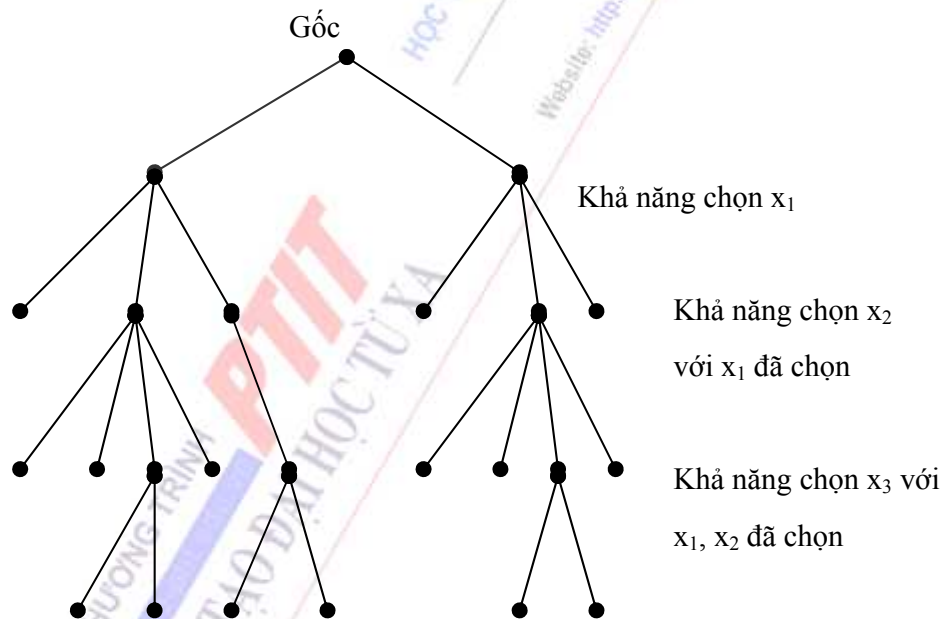
Nội dung chính của thuật toán này là xây dựng dần các thành phần của cấu hình bằng cách thử tất cả các khả năng. Giả sử cần phải tìm một cấu hình của bài toán $x = (x_1, x_2, \dots, x_n)$ mà $i-1$ thành phần x_1, x_2, \dots, x_{i-1} đã được xác định, bây giờ ta xác định thành phần thứ i của cấu hình bằng cách duyệt tất cả các khả năng có thể có và đánh số các khả năng từ 1 đến 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 2.1. Cây liệt kê lời giải theo thuật toán quay lui.

Ví dụ: Bài toán Xếp Hậu. Liệt kê tất cả các cách xếp n quân hậu trên bàn cờ $n \times n$ 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 8×8 , 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 đến 15 .

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, \dots, 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 thỏa 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, $a_j=0$ 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 hoanghai(int);
void inloigiai(int loigiai[]);FILE *fp;
int A[N], B[D], C[D], loigiai[N];
int soloigiai =0;
void hoanghai(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);
}
}

```

2.5. 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:

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

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

dẫn các thành phần của phương án. Một bộ phận gồm k thành phần (a_1, a_2, \dots, 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 thỏa mãn bất đẳng thức sau:

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

với mọi lời giải bộ phận (a_1, a_2, \dots, a_k) , và với mọi $k = 1, 2, \dots$

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

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

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

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

$\bar{f} < g(a_1, a_2, \dots, a_k) \leq \min \{ f(x) : x \in D, x_i = a_i, i = 1, 2, \dots, k \}$, vì thế tập con các phương án của bài toán $D(a_1, a_2, \dots, a_k)$ 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, \dots, a_k) . Nói cách khác, ta có thể loại bỏ các phương án trong tập $D(a_1, a_2, \dots, a_k)$ khỏi quá trình tìm kiếm.

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

```
void Try(int 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*)
    for ak ∈ Ak {
        if ( chấp nhận ak ) {
            xk = ak;
            if (k== n) < cập nhật kỷ lục>;
            else if (g(a1, a2, ..., ak) ≤  $\bar{f}$  ) Try(k+1);
        }
    }
}
```

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

```
void Nhanh_Can(){
     $\bar{f} = +\infty;$ 

    (* Nếu biết một phương án  $\bar{x}$  nào đó thì có thể đặt  $\bar{f} = f(\bar{x})$  *)

    Try(1);
    if(  $\bar{f} \leq +\infty$  ) <  $\bar{f}$  là giá trị tối ưu ,  $\bar{x}$  là phương án tối ưu >;
    else < bài toán không có phương án >;
}
```

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

```
if (k== n) < cập nhật kỷ lục >;
else if (g(a1, a2, . . . , ak) ≤  $\bar{f}$  ) Try(k+1);
```

bởi

```
if (k == n) < 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 $g(a_1, a_2, \dots, a_k)$ phải sát với giá trị của hàm mục tiêu.

Ví dụ. Giải bài toán người du lịch bằng thuật toán nhánh cận

Bài toán Người du lịch. Một người du lịch muốn đi thăm quan n thành phố T_1, T_2, \dots, 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 ($i, j = 1, 2, \dots, n$), 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:

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

với điều kiện

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

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

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

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.

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

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, \dots, u_k) có thể được tính theo công thức

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

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 2.2.

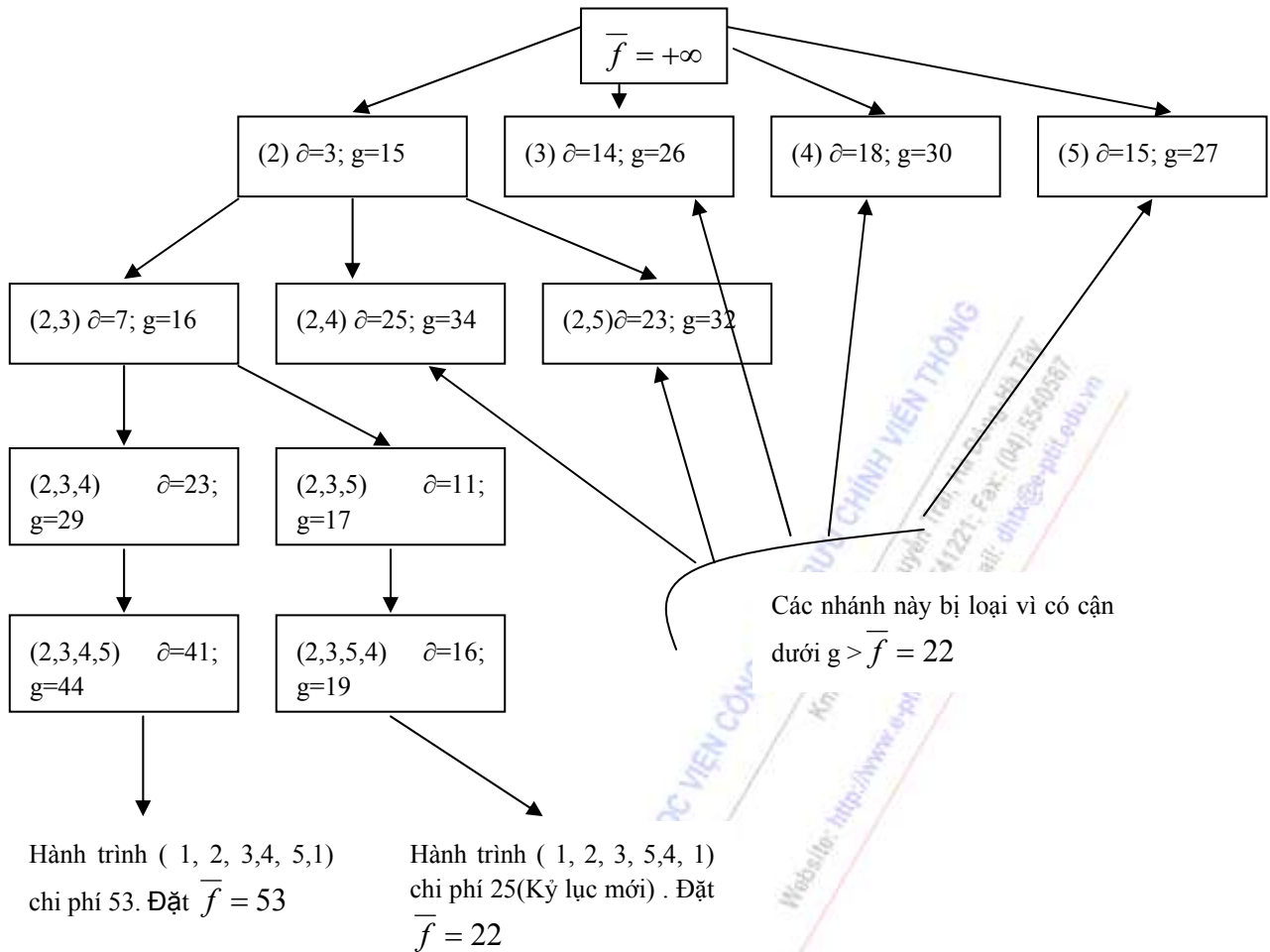
Thông tin về một phương án bộ phận trên cây được ghi trong các ô trên hình vẽ tương ứng theo thứ tự sau:

- 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 2.2. 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 So thanh pho: %d", n);
    printf("\n Ma tran chi phi:");
```

```

    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 Hanh trinh toi uu %d:", fopt);
    printf("\n Hanh trinh:");
    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]];
}

```

HỌC VIỆN CÔNG NGHỆ BƯU CHÍNH VIỄN THÔNG

Km10 Đường Nguyễn Trãi, Hà Đông-Hà Tây
 Tel: (04) 5541221; Fax: (04) 5540587
 Website: <http://www.c-pht.edu.vn>; E-mail: dhk@cpht.edu.vn

CHƯƠNG TRÌNH ĐÀO TẠO ĐẠI HỌC TỪ XA

```
        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();
}
```

HỌC VIỆN CÔNG NGHỆ BƯU CHÍNH VIỄN THÔNG

Km10 Đường Nguyễn Trãi, Hà Đông-Hà Tây
Tel: (04) 5541 221; Fax: (04) 5540 587

Website: <http://www.c-ptit.edu.vn>; E-mail: dhk@ptit.edu.vn

CHƯƠNG TRÌNH PTIT
ĐÀO TẠO ĐẠI HỌC TỪ XA

NHỮNG NỘI DUNG CẦN GHI NHỚ

- ✓ Khi không còn cách nào để giải quyết vấn đề thì có thể sử dụng cách duyệt để giải quyết.
- ✓ Tuy phương pháp định nghĩa bằng đệ qui & giải thuật đệ qui tương đối ngắn gọn và dễ hiểu nhưng không nên quá lạm dụng nó trong khi viết chương trình.
- ✓ Cần phải hiểu rõ khi nào thì phép sinh kế tiếp mới được áp dụng.
- ✓ Quá trình quay lui chỉ thực sự đúng khi ta kiểm soát được các bước trước đó.
- ✓ Để hạn chế các phép duyệt nên sử dụng phương pháp nhánh cận (nếu có thể).



HỌC VIỆN CÔNG NGHỆ BƯU CHÍNH ĐIỆN TỬ
Km10 Đường Nguyễn
Tel: (04) 5541221 Fax: (04) 5540587
Webs/le: <http://www.e-ptit.edu.vn> E-mail: info@e-ptit.edu.vn

BÀI TẬP CHƯƠNG 2

Bài 1. Duyệt mọi tập con của tập hợp 1, 2, . . . , n. Dữ liệu vào cho bởi file tapcon.in, kết quả ghi lại trong file bai1.out. Ví dụ sau sẽ minh họa cho file tapcon.in và tapcon.out.

tapcon.in	tapcon.out
3	1
	2
	2 1
	3
	3 1
	3 2
	3 2 1

Bài 2. Tìm tập con dài nhất có thứ tự tăng dần, giảm dần. Cho dãy số a_1, a_2, \dots, a_n . Hãy tìm dãy con dài nhất được sắp xếp theo thứ tự tăng hoặc giảm dần. Dữ liệu vào cho bởi file 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 file tapcon.out. Ví dụ sau sẽ minh họa cho file tapcon.in và tapcon.out.

tapcon.in	tapcon.out
5	5
7 1 3 8 9 6 12	1 3 8 9 12

Bài 3. Duyệt các tập con thỏa mãn điều kiện. Cho dãy số a_1, a_2, \dots, a_n và số M . Hãy tìm tất cả các dãy con dãy con trong dãy số a_1, a_2, \dots, a_n sao cho tổng các phần tử trong dãy con đúng bằng M . Dữ liệu vào cho bởi file 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 file tapcon.out. Ví dụ sau sẽ minh họa cho file tapcon.in và tapcon.out

tapcon.in	tapcon.out
7 50	20 30
5 10 15 20 25 30 35	

15	35		
10	15	25	
5	20	25	
5	15	30	
5	10	35	
5	10	15	20

Bài 4. Cho lưới hình chữ nhật gồm $(n \times m)$ 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 file bai14.inp, kết quả ghi lại trong file bai14.out. Ví dụ sau sẽ minh họa cho file 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
    
```

Bài 5. Duyệt mọi tập con k phần tử từ tập gồm n phần tử. Dữ liệu vào cho bởi file tapcon.in, kết quả ghi lại trong file 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
    
```

Bài 6. Duyệt các tập con k phần tử thỏa mãn điều kiện. Cho dãy số a_1, a_2, \dots, a_n và số M . Hãy tìm tất cả các dãy con k phần tử trong dãy số a_1, a_2, \dots, a_n sao cho tổng các phần tử trong dãy con đúng bằng M . Dữ liệu vào cho bởi file 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 file tapcon.out. Ví dụ sau sẽ minh họa cho file 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
    
```

Bài 7. Duyệt mọi hoán vị của từ COMPUTER. Dữ liệu vào cho bởi file hoanvi.in, kết quả ghi lại trong file hoanvi.out.

Bài 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 $1, 2, \dots, n$ vào các hình vuông đơn vị sao cho những điều kiện sau được thỏa mãn:

- Đọc theo hàng ta nhận được n hoán vị khác nhau của $1, 2, \dots, n$;
- Đọc theo cột ta nhận được n hoán vị khác nhau của $1, 2, \dots, n$;
- Đọc theo hai đường chéo ta nhận được 2 hoán vị khác nhau của $1, 2, \dots, n$;

Hãy tìm ít nhất 1 (hoặc tất cả) các hình vuông thỏa mãn 3 điều kiện trên. Dữ liệu vào cho bởi file hoanvi.in, kết quả ghi lại trong file hoanvi.out. Ví dụ sau sẽ minh họa cho file 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
    
```

Bài 9. Duyệt mọi cách chia số tự nhiên n thành tổng các số nguyên nhỏ hơn. Dữ liệu vào cho bởi file chiaso.in, kết quả ghi lại trong file chiaso.out. Ví dụ sau sẽ minh họa cho file input & output của bài toán.

chiaso.in

4

chiaso.out

4

3 1

2 2

2 1 1

1 1 1 1

Bài 10. Duyệt mọi bộ giá trị trong tập các giá trị rời rạc. Cho k tập hợp các số thực $A_1, A_2, \dots, A_k (k \leq 10)$ có số các phần tử tương ứng là N_1, N_2, \dots, N_k (các tập có thể có những phần tử giống nhau). Hãy duyệt tất cả các bộ k phần tử $a=(a_1, a_2, \dots, a_k)$ sao cho $a_i \in A_i (i=1, 2, \dots, k)$. Dữ liệu vào cho bởi file 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, \dots, N_k ; k dòng kế tiếp ghi lại các phần tử của tập hợp A_1, A_2, \dots, A_k . Kết quả ghi lại trong file chiaso.out, mỗi phần tử được phân biệt với nhau bởi một vài dấu trống. Ví dụ sau sẽ minh họa cho file 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

Bài 11. Tìm bộ giá trị rời rạc trong bài 21 để hàm mục tiêu $\sin(x_1+x_2 + \dots + x_k)$ đạt giá trị lớn nhất. Dữ liệu vào cho bởi file bai22.inp, kết quả ghi lại trong file bai22.out.

Bài 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 ? 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.

Bài 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 ($1 \leq i \leq n$). 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 file 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 file 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 file input & output của bài toán.

caitui.in

4	8		
10	5	3	6
5	3	2	4

caitui.out

15			
1	1	0	0
1	1	0	0

Bài 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 ($1 \leq i \leq n$). 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 file `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 file `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 họa cho file input & output của bài toán.

`caitui.in`

```
4      8
8      5      3      1
4      3      2      1
```

`caitui.out`

```
14
1      1      0      1
```

Bài 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 file `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 file `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 file 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 3: NGĂN XẾP, HÀNG ĐỢI VÀ DANH SÁCH MÓC NỐI (STACK, QUEUE, LINK LIST)

Nội dung chính của chương này nhằm làm rõ các phương pháp, kỹ thuật biểu diễn, phép toán và ứng dụng của các cấu trúc dữ liệu trừu tượng. Cần đặc biệt lưu ý, ứng dụng các cấu trúc dữ liệu này không chỉ riêng cho lập trình ứng dụng mà còn ứng dụng trong biểu diễn bộ nhớ để giải quyết những vấn đề bên trong của các hệ điều hành. Các kỹ thuật lập trình trên cấu trúc dữ liệu trừu tượng được đề cập ở đây bao gồm:

- ✓ Kỹ thuật lập trình trên ngăn xếp.
- ✓ Kỹ thuật lập trình trên hàng đợi.
- ✓ Kỹ thuật lập trình trên danh sách liên kết đơn.
- ✓ Kỹ thuật lập trình trên danh sách liên kết kép.

Bạn đọc có thể tìm thấy những cài đặt và ứng dụng cụ thể trong tài liệu [1].

3.1. KIỂU DỮ LIỆU NGĂN XẾP VÀ ỨNG DỤNG

3.1.1. Định nghĩa và khai báo

Ngăn xếp (Stack) hay bộ xếp chồng là một kiểu danh sách tuyến tính đặc biệt mà phép bổ xung phần tử và loại bỏ phần tử luôn luôn được thực hiện ở một đầu gọi là đỉnh (top).

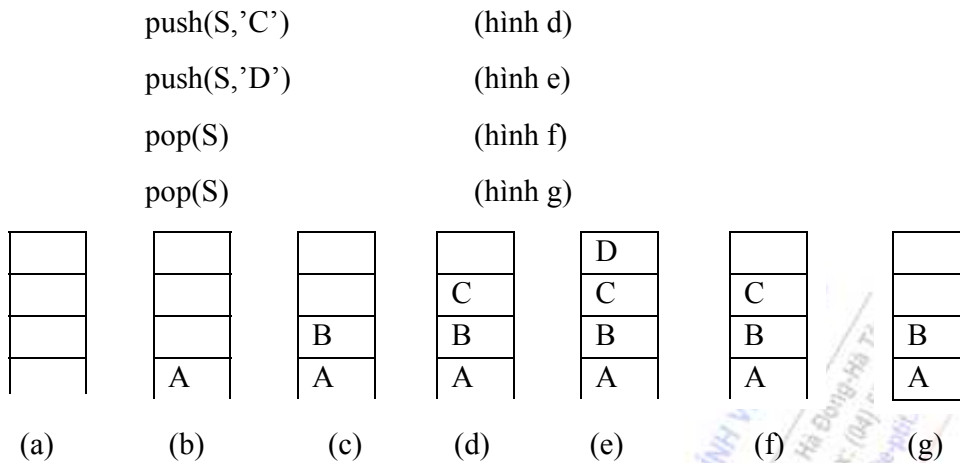
Có thể hình dung stack như một chồng đĩa được xếp vào hộp hoặc một băng đạn được nạp vào khẩu súng liên thanh. Quá trình xếp đĩa hoặc nạp đạn chỉ được thực hiện ở một đầu, chiếc đĩa hoặc viên đạn cuối cùng lại chiếm vị trí ở đỉnh đầu tiên còn đĩa đầu hoặc viên đạn đầu lại ở đáy của hộp (bottom), khi 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 trên stack là thêm một nút vào đỉnh stack (push) và loại bỏ một nút tại đỉnh stack (pop). Khi muốn thêm một nút vào 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 a là trạng thái rỗng, hình e mô tả trạng thái đầy. Các thao tác:

push(S, 'A') (hình b)

push(S, 'B') (hình c)



Hình 3.1. Các thao tác trên Stack

Có thể lưu trữ stack dưới dạng một vector S gồm n thành phần liên tiếp nhau. Nếu T 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.



Hình 3.2. Vector S lưu trữ Stack

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;
    
```

3.1.2. 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 --]);
}
```

3.1.3. Ứng dụng của stack

Stack được ứng dụng để biểu diễn nhiều thuật giải phức tạp khác nhau, đặc biệt đối với những bài toán cần sử dụng đến các lời gọi đệ qui. Dưới đây là một số các ví dụ điển hình của việc ứng dụng stack.

Đảo ngược chuỗi ký tự: Quá trình đảo ngược một chuỗi ký tự giống như việc đưa vào (push) từng ký tự trong chuỗi vào stack, sau đó đưa ra (pop) các ký tự trong stack ra cho tới khi stack rỗng ta được một chuỗi đảo ngược.

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.

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 lũy thừa rồi đẩy vào stack. Cuối cùng ta nhận được $25^2 = 625$.

Dưới đây là chương trình đảo ngược chuỗi kí tự sử dụng stack. Những ví dụ khác, bạn đọc có thể tìm thấy trong các tài liệu [1], [2].

Ví dụ 3.1. Chương trình đảo ngược chuỗi kí tự.

```
#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 Nhap String:");gets(chuoi);
        vitri=strlen(chuoi);
        for (i=0; i<vitri;i++)
            Push(&s, chuoi[i]);
        while(!Empty(&s))
            printf("%c", Pop(&s));
        getch();
    }

```

3.2. HÀNG ĐỢI (QUEUE)

3.2.1. Định nghĩa và khai báo

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).

Ta có thể khai báo hàng đợi như một danh sách tuyến tính gồm MAX phần tử mỗi phần tử là một cấu trúc, hai biến front, rear trở lối vào và lối ra trong queue. Ví dụ dưới đây định nghĩa một hàng đợi của các sản phẩm gồm hai thuộc tính mã hàng (mahang) và tên hàng (ten).

```

typedef struct{
    int mahang;

```



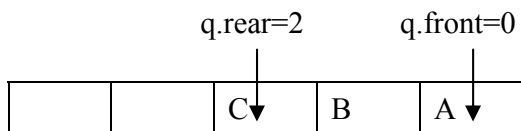
```

char ten[20];
} hang;
typedef struct {
    int front, rear;
    hang node[MAX];
} q;
    
```

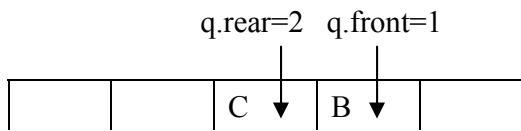
Để truy nhập vào hàng đợi, chúng ta sử dụng hai biến con trỏ *front* chỉ lối trước và *rear* chỉ lối sau. Khi lối trước trùng với lối sau ($q.rear = q.front$) thì *queue* ở trạng thái rỗng (hình a), để thêm dữ liệu vào hàng đợi các phần tử *A*, *B*, *C* được thực hiện thông qua thao tác *insert(q,A)*, *insert(q,B)*, *insert(q,C)* được mô tả ở hình b, thao tác loại bỏ phần tử khỏi hàng đợi *Remove(q)* được mô tả ở hình c, những thao tác tiếp theo được mô tả tại hình d, e.



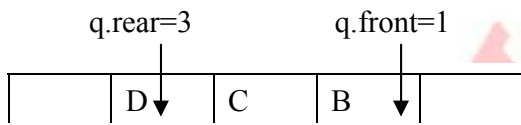
Hình a. Trạng thái rỗng của hàng đợi.



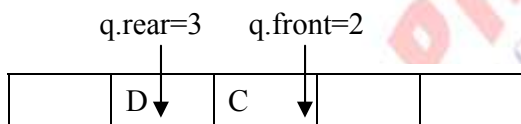
Hình b. *insert(Q,A)*; *insert(Q,B)*, *insert(Q,C)*



Hình c. *remove(q)*.



Hình d. *insert(q,D)*.



Hình e. *remove(q)*.

Hình 3.3. Các thao tác trên Hàng đợi (Queue)

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. 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.

3.2.2. 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. Bạn đọc có thể tham khảo thêm trong các tài liệu [1], [2]. Dưới đây, chúng ta đưa ra một ứng dụng của hàng đợi để giải quyết bài toán “Nhà sản xuất và Người tiêu dùng”.

Ví dụ 3.2- Giải quyết bài toán ”*Người sản xuất và nhà tiêu dùng* “ với số các vùng đệm hạn chế.

Chúng ta 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 chỉ được phép sử dụng trong số n mặt hàng. Tuy nhiên, người sản xuất chỉ có thể lưu trữ vào kho khi và chỉ khi kho chưa bị đầy. Ngược lại, 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:

Ta xây dựng hàng đợi như một danh sách tuyến tính gồm MAX phần tử mỗi phần tử là một cấu trúc, hai biến *front*, *rear* trỏ đến 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, khi đó 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);
}

```

Dưới đây là toàn bộ văn bản 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 NGUOI SAN XUAT/ NHA TIEU DUNG");
        printf("\n 1- Nhap mot mat hang");
        printf("\n 2- Xuat mot mat hang");
        printf("\n 3- Xem mot mat hang");
        printf("\n 4- Xem hang moi nhap");
        printf("\n 5- Xem tat ca");
        printf("\n 6- Xuat toan bo");
        printf("\n Chuc nang chon:");chucnang=getch();
        switch(chucnang){
            case '1':
                printf("\n Ma mat hang:"); scanf("%d", &mh.mahang);
                printf("\n Ten hang:");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)?q.front+1;
                printf("\n Hang xuat");
                printf("\n%6d%20s",q.node[front1].mahang,q.node[front1].ten);
                break;
```

```

case '4':
    printf("\n Hang moi nhap");
    printf("\n%5d%20s",q.node[q.rear].mahang,q.node[q.rear].ten);
    break;
case '5':
    printf("\ Hang trong kho");
    Traverse(&q);delay(2000);break;
    }
} while(chucnang!='0');
}

```

3.3. DANH SÁCH LIÊN KẾT ĐƠN

3.3.1. 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.

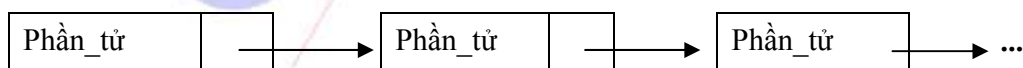
Danh sách móc nối đơn giản dưới đây xây dựng mỗi đỉnh của danh sách chỉ lưu giữ một biến nguyên.

```

/*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 3.4. 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*tiếp;
};
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 3.4, ngoại trừ việc mỗi *phantu* là một biến nguyên.

3.3.2. Các thao tác trên danh sách móc nối

Các thao tác trên danh sách móc nối bao gồm việc cấp phát bộ nhớ cho các đỉnh và gán dữ liệu cho con trỏ. Để danh sách được tạo nên đúng đắn, ta biểu diễn 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);
}
```

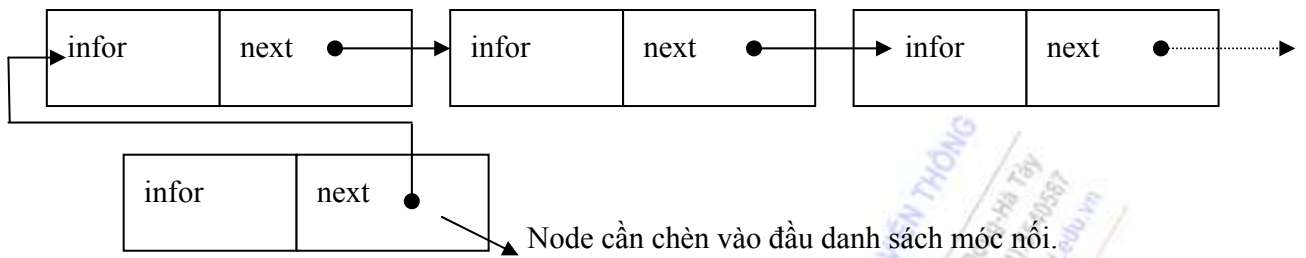
Chèn một phần tử mới vào đầu danh sách:

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 trên hình 3.5.



Hình 3.5. Thêm đỉnh mới vào đầu danh sách móc nối đơn

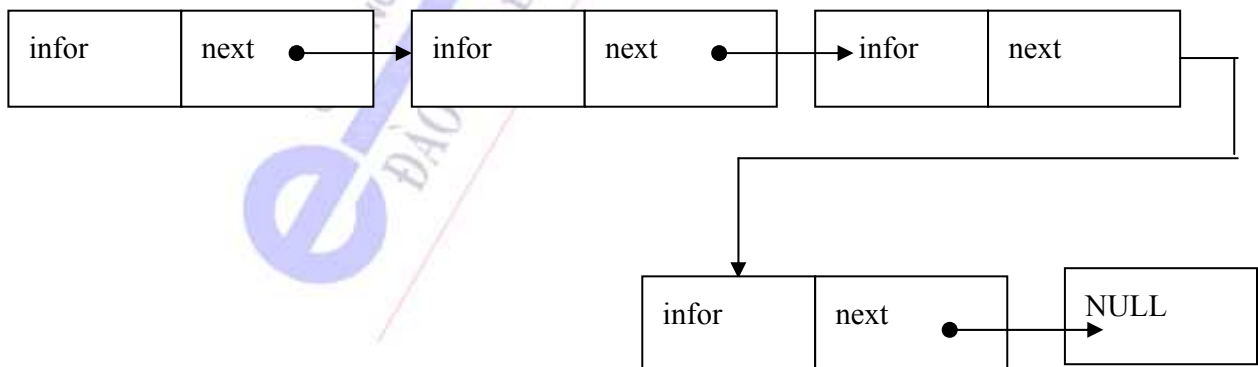
```
void Push_Top( NODEPTR *plist, int x) {
    NODEPTR p;
    p= Getnode(); // cấp không gian nhớ cho đỉnh mới
    p -> infor = x; // gán giá trị thích hợp cho đỉnh mới
    p ->next = *plist;
    *plist = p; // thiết lập liên kết
}
```

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 3.6



Hình 3.6. 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
}
}
```

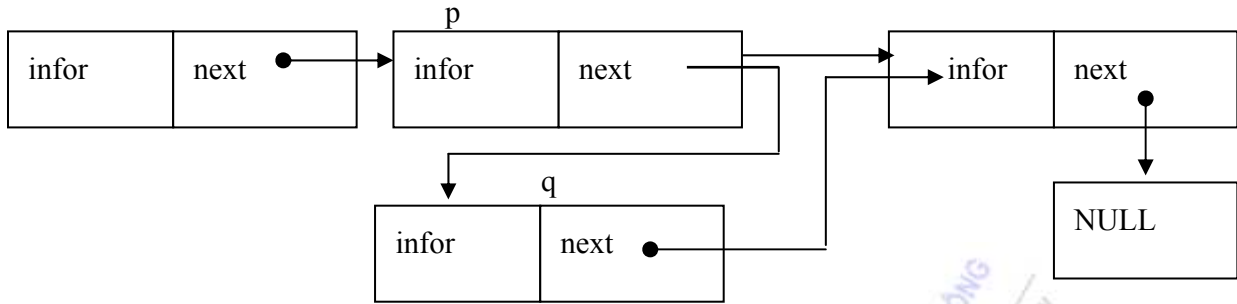
Thêm node mới q vào giữa danh sách trước node p :

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 $p->next \neq NULL$. 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 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 3.7. 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 thể thực hiện việc loại bỏ. Trong trường hợp còn lại, ta thực hiện như sau:

- ✓ Dùng node p trở 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 trở 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);
}

```

Bạn đọc có thể tìm thấy những cài đặt cụ thể của danh sách liên kết đơn trong các tài liệu [1], [2].

3.4. 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 3.8. 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 lập liên kết trái
    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
        //chuyển con trỏ tới node cuối danh sách
    }
}

```

```
q = *plist;
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 trở 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
}
```



HỆ VIỆN CÔNG NGHỆ THÔNG TIN CHÍNH VIÊN THÔNG
Km10 Đường Nguyễn Trãi, Hà Đông-Hà Tây
Tel: (04) 5541221; Fax: (04) 5540587
Website: <http://www.e-ptit.edu.vn>; E-mail: dhk@e-ptit.edu.vn

NHỮNG NỘI DUNG CẦN GHI NHỚ

- ✓ Các phương pháp định nghĩa stack, khi nào dùng stack & vai trò của stack đối với các giải thuật đệ qui.
- ✓ Phương pháp định nghĩa hàng đợi, các thao tác trên hàng đợi và ứng dụng của hàng đợi.
- ✓ Bản chất động là tính chất cơ bản nhất của danh sách liên kết đơn và liên kết kép.
- ✓ Sự khác biệt cơ bản của danh sách liên kết đơn và danh sách liên kết kép là các con trỏ left và right.
- ✓ Những ứng dụng lớn thường được cài đặt trên các cấu trúc dữ liệu động.
- ✓ Chú ý giải phóng bộ nhớ cho con trỏ trong khi lập trình.



HỌC VIỆN CÔNG NGHỆ BƯỞI CHÍNH VIỆN THÔNG
Km10 Đường Nguyễn Trãi, Hà Nội, Việt Nam
Tel: (04) 5540 2111 Fax: (04) 5540 2107
Website: <http://www.e-ptit.edu.vn> E-mail: hnk@ptit.edu.vn

BÀI TẬP CHƯƠNG 3

Bài 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 File `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
    
```

Bài 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;
- Hiện 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;
    
```

Bài 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:

- Quy 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.

- Quy 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 e1 và e2 là hai biểu thức có dạng viết Ba Lan tương ứng là d1 và d2 thì dạng viết Ba Lan của $e1 + e2$ là $d1 d2+$, của $e1 - e2$ là $d1 d2-$, của $e1 * e2$ là $d1 d2*$ (Giữa d1 và d2 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 file 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 file balan.out theo từng dòng. Ví dụ: với file 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*-

Bài 4. Tính toán giá trị biểu thức Ba Lan. Cho file 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 file 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 file 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

Bài 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 thỏa 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 file dữ liệu lịch.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 file lịch.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 file lịch.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_Action3 2
6 Writing_Data 2 20
7 Reading_Data 2 10
    
```

lich.out

3 4 5 7 6 1 2

63.5

Bài 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.



HỌC VIỆN CÔNG NGHỆ BƯU CHÍNH ĐIỆN TỬ
Km10 Đường Nguyễn
Tel: (04) 554 1111
Website: <http://www.e-ptit.edu.vn> E-mail: info@e-ptit.edu.vn

CHƯƠNG 4: CẤU TRÚC DỮ LIỆU CÂY (TREE)

Cây là một trong những cấu trúc dữ liệu rời rạc có ứng dụng quan trọng trong biểu diễn tính toán, biểu diễn tri thức & biểu diễn các đối tượng dữ liệu phức tạp. Trọng tâm chính của chương này nhằm cung cấp cho bạn đọc những khái niệm và thao tác cơ bản trên cây nhị phân, bao gồm:

- ✓ Khái niệm về cây, cây nhị phân, cây nhị phân tìm kiếm.
- ✓ Khái niệm node gốc (root), node lá (leaf), mức (level) & độ sâu của cây.
- ✓ Phương pháp biểu diễn và các thao tác trên cây nhị phân.
- ✓ Các thao tác duyệt cây: duyệt theo thứ tự trước, duyệt theo thứ tự giữa & duyệt theo thứ tự sau.
- ✓ Phương pháp biểu diễn và các thao tác trên cây nhị phân tìm kiếm.

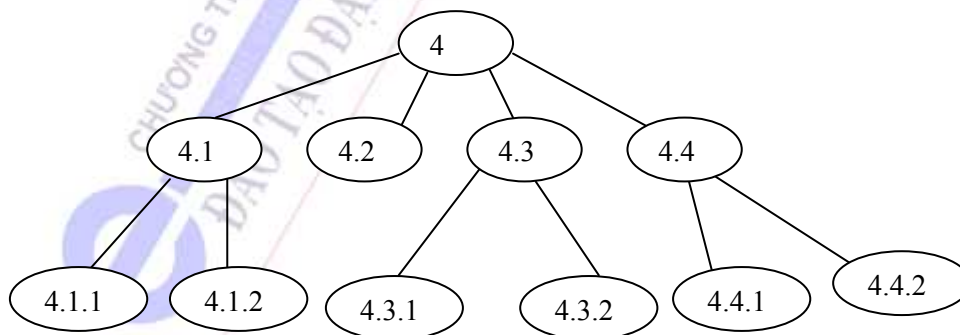
Bạn đọc có thể tìm hiểu sâu hơn về cây nhiều nhánh, cây cân bằng và cây nhị phân hoàn toàn cân bằng trong tài liệu [1].

4.1. 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:

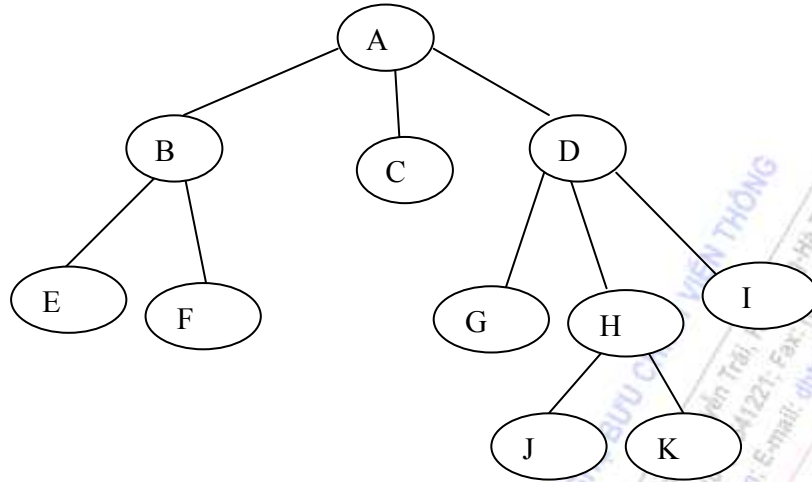
- Một node là một cây. Node đó cũng là gốc (root) của cây ấy.
- Nếu n là một node và T_1, T_2, \dots, T_k là các cây với n_1, n_2, \dots, n_k lần lượt là gốc thì một cây mới T sẽ được tạo lập bằng cách cho node n trở thành cha của các node n_1, n_2, \dots, n_k hay node n trở thành gốc và T_1, T_2, \dots, 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 4.1. 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 4.2 sau, 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 4.2. mô tả cấp của cây

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 trong hình 4.2 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.

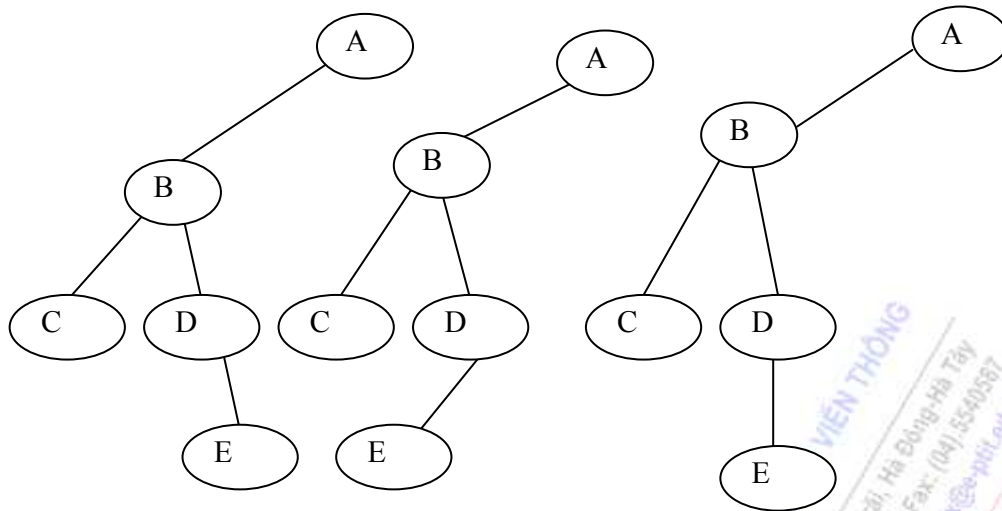
Chiều cao (height) hay chiều sâu (depth) của một cây là số mức lớn nhất của node trên cây đó. Cây 4.2 có chiều cao là 4.

Đường đi từ node n_1 đến n_k là dãy các node n_1, n_2, \dots, n_k sao cho n_i là node cha của node n_{i+1} ($1 \leq i < k$), 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 4.2 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.

4.2. CÂY NHỊ PHÂN

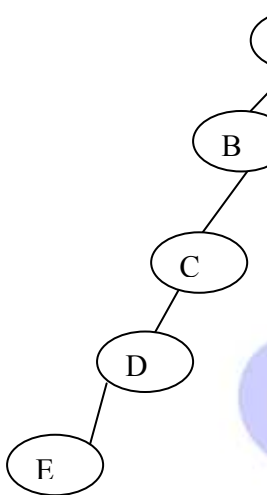
Cây nhị phân là một dạng quan trọng của cấu trúc cây có đặc điểm là mọi node trên cây chỉ có tối đa là hai node con. Cây con bên trái của cây nhị phân được gọi là *left subtree*, cây con bên phải của cây được gọi là *right subtree*. Đối với cây nhị phân, bao giờ cũng được phân biệt cây con bên trái và cây con bên phải. Như vậy, cây nhị phân là một cây có thứ tự. Ví dụ trong hình 4.3 đều là các cây nhị phân:



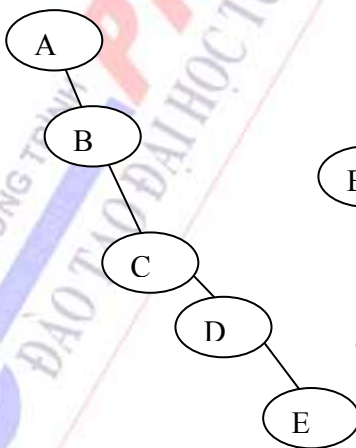
Hình 4.3. Cây nhị phân

Các cây nhị phân có dạng đặc biệt bao gồm:

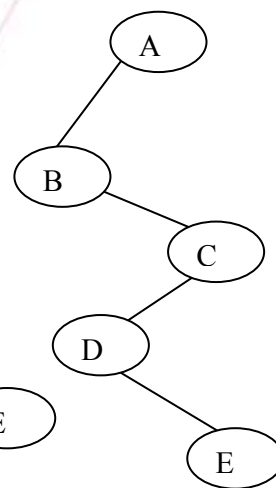
- **Cây nhị phân lệch trái** (hình 4.4a): là cây nhị phân chỉ có các node bên trái.
- **Cây nhị phân lệch phải** (hình 4.4b): là cây chỉ bao gồm các node phải.
- **Cây nhị phân zig zắc** (hình 4.4 c, 4.4d): node trái và node phải của cây đan xen nhau thành một hình zig zắc.
- **Cây nhị phân hoàn chỉnh** (strictly binary tree: hình 4.4e) : 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 4.4f): 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 .



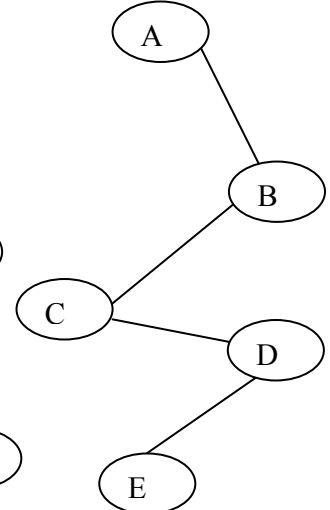
Hình 4.4a



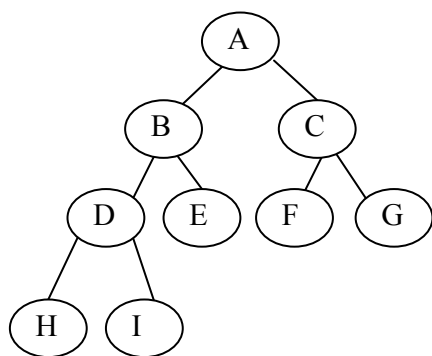
Hình 4.4b



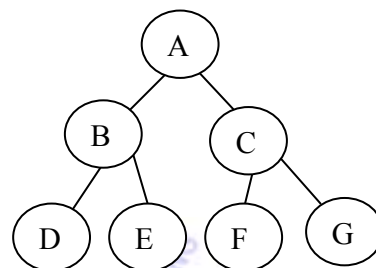
Hình 4.4c



Hình 4.4d



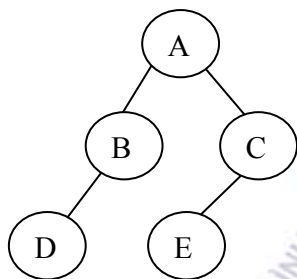
Hình 4.4 e



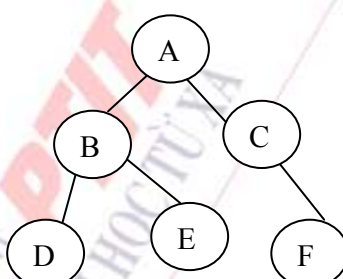
Hình 4.4 f

Cây nhị phân hoàn toàn cân bằng (hình 4.5): 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 lệch 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 là một trong 3 trường hợp:

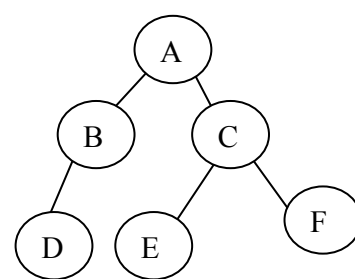
- Số node nhánh cây con bên trái bằng số node nhánh cây con bên phải bằng ($N_l = N_r$) (hình 4.5a).
- Số node nhánh cây con bên trái bằng số node nhánh cây con bên phải cộng 1 ($N_l = N_r + 1$) (hình 4.5b)
- Số node nhánh cây con bên trái bằng số node nhánh cây con bên phải trừ 1 ($N_l = N_r - 1$) (hình 4.5c).



Hình 4.5a



Hình 4.5b

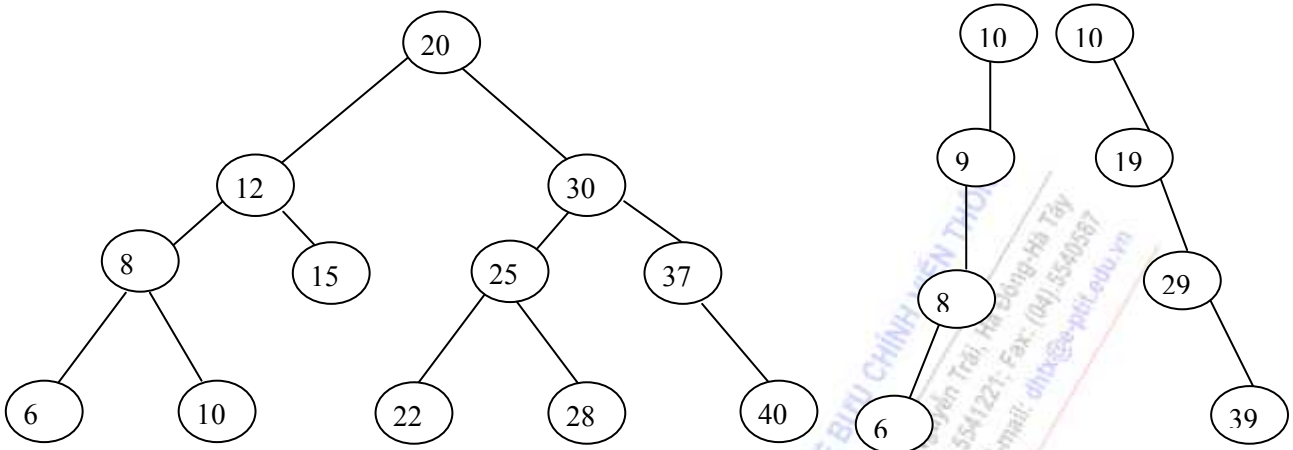


Hình 4.5c

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.



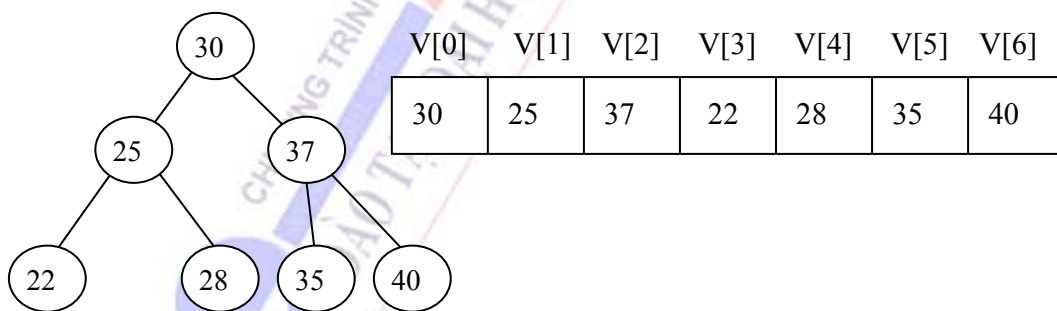
Hình 4.6. Ví dụ về cây nhị phân tìm kiếm

4.3. BIỂU DIỄN CÂY NHỊ PHÂN

4.3.1. Biểu diễn cây nhị phân bằng danh sách tuyến tính

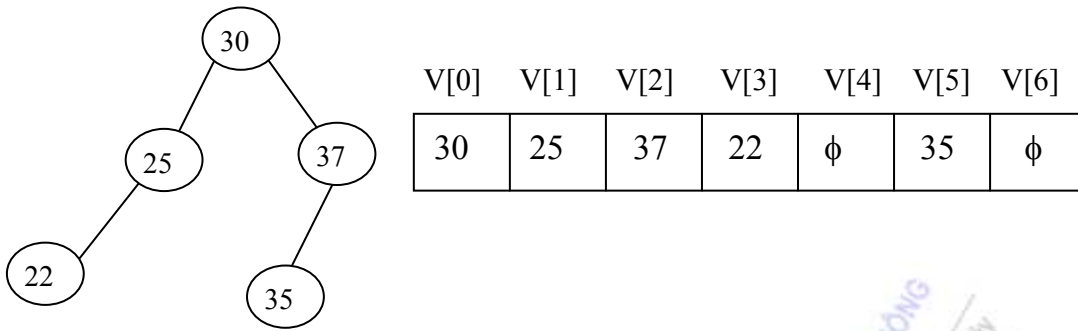
Trong trường hợp cây nhị phân đầy đủ, ta có thể dễ dàng biểu diễn cây nhị phân bằng một mảng lưu trữ kế tiếp. Trong đó node gốc là phần tử đầu tiên của mảng (phần tử thứ 1), node con thứ $i >= 1$ của cây nhị phân là phần tử thứ $2i$, $2i + 1$ hay cha của node thứ j là $[j/2]$. Với qui tắc đó, cây nhị 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 trong hình 4.7 sẽ được lưu trữ kế tiếp như sau:



Hình 4.7. 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 như trong ví dụ sau:



Hình 4.8- Lưu trữ kế tiếp của cây nhị phân không đầy đủ

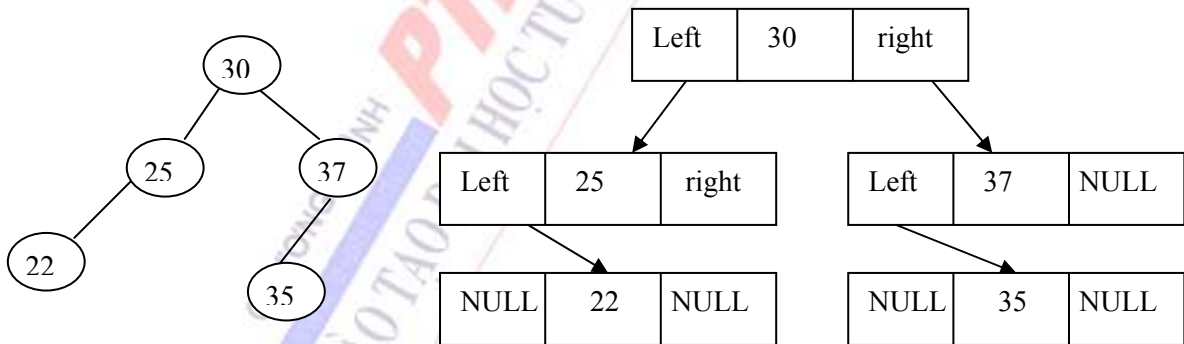
4.3.2. Biểu diễn cây nhị phân bằng danh sách móc nối

Trong cách lưu trữ cây nhị phân bằng danh sách móc nối, mỗi node được mô tả bằng ba loại thông tin chính : left 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 4.9.



Hình 4.9. mô tả một node của cây nhị phân.

Ví dụ: cây nhị phân trong hình 4.10 sẽ được biểu diễn bằng danh sách liên kết như sau:



Hình 4.10. Biểu diễn cây nhị phân bằng danh sách móc nối .

4.4. CÁC THAO TÁC TRÊN CÂY NHỊ PHÂN

4.4.1. 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];
```

4.4.2. Định nghĩa cây nhị phân theo danh sách liên kết:

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

4.4.3. 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 make node đã được xây dựng như trên;

```
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 ($p == \text{NULL}$), thì ta không thể thực hiện được thao tác thêm node lá vào node phải node p;
- Nếu node p có thực ($p \neq \text{NULL}$) 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();

```
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 ($p \neq \text{NULL}$) 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 ($p \neq \text{NULL}$) 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 ($q ->left \neq \text{NULL} \parallel q ->right \neq \text{NULL}$);
 - Nếu node q là node lá ($q ->left == \text{NULL} \ \&\& \ q ->right == \text{NULL}$) 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);
}

```



```

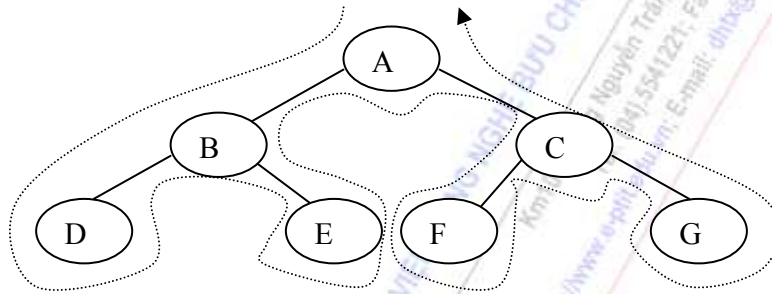
return(p);
}

```

4.5. CÁC PHÉP DUYỆT CÂY NHỊ PHÂN (TRAVERSING BINARY TREE)

Phép duyệt cây là phương pháp viếng thăm (visit) các node một cách có hệ thống sao cho mỗi node chỉ được thăm đúng 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 4.11. mô tả phương pháp duyệt cây nhị phân

4.5.1. 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 4.11 thì phép duyệt Preorder cho ta kết quả duyệt theo thứ tự các node là :A -> B -> D -> E -> C -> F -> G.

Với phương pháp duyệt theo thứ tự trước, chúng ta có thể cài đặt cho cây được định nghĩa trong mục 4.4 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
        Pretraverse(proot ->left); // duyệt nhánh cây con bên trái
        Pretraverse(proot ->right); // Duyệt nhánh con bên phải
    }
}

```

4.5.2. 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 4.11 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 4.4 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
    }
}
```

4.5.3. 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 4.11 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 4.4 bằng một thủ tục đệ qui như sau:

```
void Posttravese ( 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
    }
}
```

4.6. CÀI ĐẶT CÂY NHỊ PHÂN TÌM KIẾM

Những cài đặt cụ thể cho cây nhị phân và cây nhị phân đầy đủ đã được trình bày trong [1]. Dưới đây là một cài đặt cụ thể cho cây nhị phân tìm kiếm bằng danh sách móc nối.

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 $x < \text{gốc}$ và chưa có lá con bên trái thì thực hiện thêm node vào nhánh bên trái.
- Nếu $x > \text{gốc}$ 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 Nut p khong co thuc");
        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 cụ thể các thao tác trên cây nhị phân tìm kiếm được thể hiện như dưới đây.

```

#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 thuc");
    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 thuc");
    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 Nut p không có thực");
        delay(2000);return(p);
    }
    if(p->right==NULL)
        rp=p->left;
    else {
        if (p->left==NULL)
            rp = p->right;
        else {
```

HỌC VIỆN CÔNG NGHỆ BƯU CHÍNH VIỄN THÔNG
Km10 Đường Nguyễn Trãi, Hà Đông-Hà Tây
Tel: (04) 5541 221; Fax: (04) 5540 587
Website: <http://www.c-ptit.edu.vn>; E-mail: dhk@ptit.edu.vn

PTIT
TRƯỜNG ĐẠI HỌC TỰ NHIÊN

```

        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 CAY NHI PHAN TIM KIEM");
        printf("\n 1-Them nut tren cay");
        printf("\n 2-Xoa node goc");
        printf("\n 3-Xoa node con ben trai");
        printf("\n 4-Xoa node con ben phai");
        printf("\n 5-Xoa toan bo cay");
        printf("\n 6-Duyet cay theo NLR");
        printf("\n 7-Duyet cay theo LNR");
        printf("\n 8-Duyet cay theo LRN");
        printf("\n 9-Tim kiem tren cay");
    }
}

```



```
printf("\n 0-Thoat khoi chuong trinh");
printf("\n Lua chon chuc nang:");
scanf("%d", &chucnang);
switch(chucnang){
    case 1:
        printf("\n Noi dung nut moi:");
        scanf("%d",&noidung);
        if(ptree==NULL)
            ptree=Makenode(noidung);
        else
            Insert(ptree,noidung);
        break;
    case 2:
        if (ptree==NULL)
            printf("\n Cay bi rong");
        else
            ptree=Remove(ptree);
        break;
    case 3:
        printf("\n Noi dung node cha:");
        scanf("%d", &noidung);
        p=Search(ptree,noidung);
        if (p!=NULL)
            p->left = Remove(p->left);
        else
            printf("\n Khong co node cha");
        break;
    case 4:
        printf("\n Noi dung node cha:");
        scanf("%d", &noidung);
        p=Search(ptree,noidung);
        if (p!=NULL)
            p->right = Remove(p->right);
        else
            printf("\n Khong co node cha");
        break;
    case 5:
        Cleartree(ptree);
        break;
    case 6:
        printf("\n Duyet cay theo NLR");
```

```
        if(ptree==NULL)
            printf("\n Cay rong");
        else
            Pretrav(ptree);
        break;
case 7:
    printf("\n Duyet cay theo LNR");
    if(ptree==NULL)
        printf("\n Cay rong");
    else
        Intrav(ptree);
    break;
case 8:
    printf("\n Duyet cay theo NRN");
    if(ptree==NULL)
        printf("\n Cay rong");
    else
        Postrav(ptree);
    break;
case 9:
    printf("\n Noi dung can tim:");
    scanf("%d",&noidung);
    if(Search(ptree,noidung))
        printf("\n Tim thay");
    else
        printf("\n Khong tim thay");
    break;
    }
    delay(1000);
} while(chucnang!=0);
Cleartree(ptree); ptree=NULL;
}
```

NHỮNG NỘI DUNG CẦN GHI NHỚ

- ✓ Định nghĩa cây, cây nhị phân, cây cân bằng và cây hoàn toàn cân bằng. Các khái niệm mức, độ sâu của cây.
- ✓ Các phương pháp duyệt cây: duyệt theo thứ tự trước, duyệt theo thứ tự giữa và duyệt theo thứ tự sau.
- ✓ Phân biệt được những thao tác giống nhau và khác nhau cây nhị phân tìm kiếm và cây nhị phân thông thường.
- ✓ Tìm hiểu thêm về cây nhiều nhánh trong các tài liệu [1], [2].
- ✓ Tìm hiểu thêm về cây quyết định và ứng dụng của nó trong biểu diễn tri thức.



HỌC VIỆN CÔNG NGHỆ BƯỞI CÂY NHỊ THON
Km10 Đường Nguyễn Trãi, Hà Đông, Hà Tây
Tel: (04) 5555 2211 Fax: (04) 5555 5587
Website: <http://www.e-ptit.edu.vn> E-mail: nhk@ptit.edu.vn

BÀI TẬP CHƯƠNG 4

Bài 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.

Bài 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 ($0 \leq i \leq d$) cây có đúng 2^i 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.

Bài 3. Một cây nhị phân được gọi là cây nhị phân gần đầy với độ sâu d nếu với mọi mức i ($0 \leq i \leq d-1$) 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?

Bài 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;

Bài 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;
- 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 ?

Bài 6. 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 & output như sau:

cay.in

```

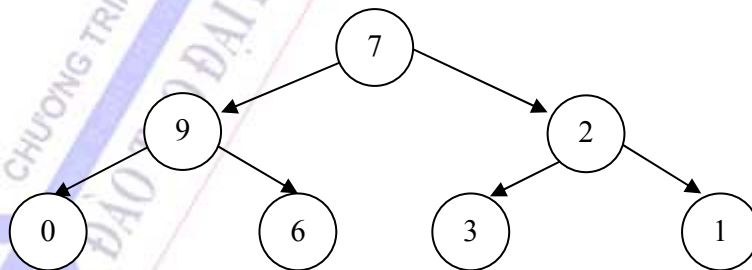
5
7
2   8
8   1   0
2   7   4   4
4   5   2   6   5
  
```

cay.out

```

30
7   3   8   7   5
  
```

Bài 7. 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:



Bài 8. 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 5: ĐỒ THỊ (GRAPH)

Đồ thị là một cấu trúc dữ liệu rời rạc nhưng lại có ứng dụng hiện đại. Đồ thị có thể dùng để biểu diễn các sơ đồ của một mạch điện, biểu diễn đường đi của hệ thống giao thông hay các loại mạng máy tính. Nắm bắt được những thuật toán trên đồ thị giúp chúng ta giải quyết được nhiều bài toán tối ưu quan trọng như bài toán qui hoạch mạng, bài toán phân luồng trên mạng hay phân luồng giao thông, bài toán tìm đường đi ngắn nhất hoặc cực tiểu hoá chi phí cho các hoạt động sản xuất kinh doanh. Những nội dung được trình bày bao gồm:

- ✓ Định nghĩa đồ thị, phân loại đồ thị và những khái niệm cơ bản liên quan.
- ✓ Các phương pháp biểu diễn đồ thị trên máy tính.
- ✓ Các thuật toán tìm kiếm trên đồ thị.
- ✓ Đồ thị Euler & đồ thị hamilton.
- ✓ Bài toán tìm cây bao trùm nhỏ nhất.
- ✓ Bài toán tìm đường đi ngắn nhất

Bạn đọc có thể tìm thấy những cài đặt cụ thể và những kiến thức sâu hơn về Lý thuyết đồ thị trong tài liệu [1] & [3].

5.1. NHỮNG KHÁI NIỆM CƠ BẢN CỦA ĐỒ THỊ

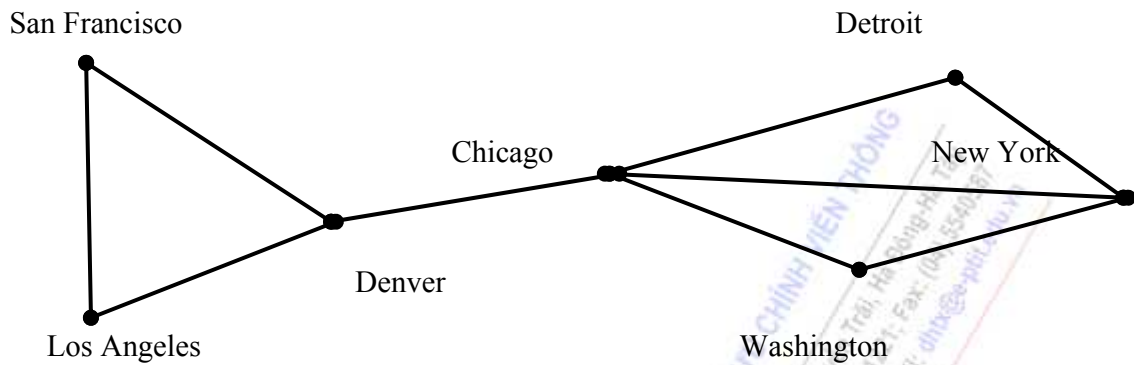
5.1.1. Các loại đồ thị

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 Thụy Sĩ Leonhard Euler đề xuất và chính ông là người dùng lý thuyết đồ thị giải quyết bài toán nổi tiếng “Cầu Königsberg”.

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 5.1 là sơ đồ của mạng máy tính loại 1.

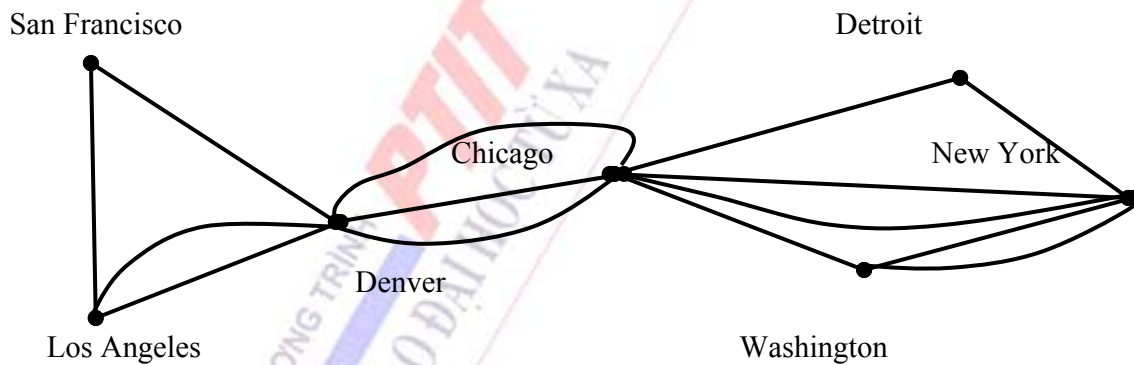


Hình 5.1. Mạng máy tính đơn kênh thoại.

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**.

Định nghĩa 1. Đơn đồ thị vô hướng $G = \langle V, E \rangle$ 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 5.2.



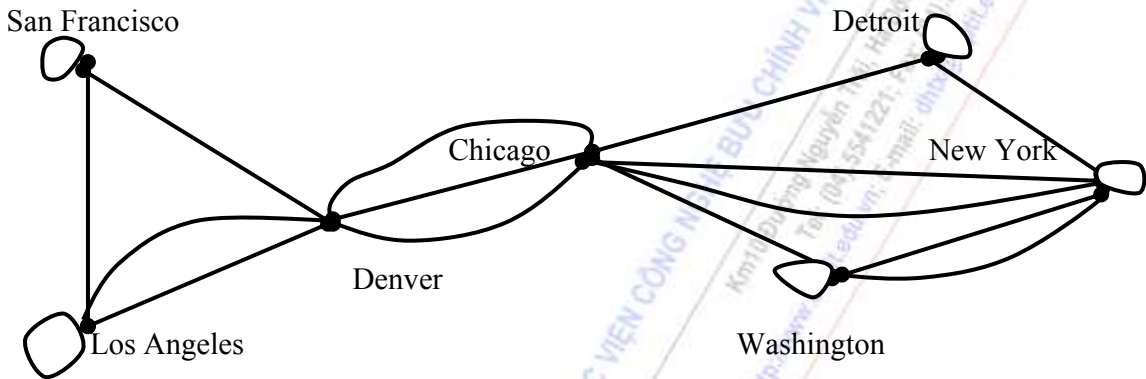
Hình 5.2. Mạng máy tính đa kênh thoại.

Trên hình 5.2, 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 $G = \langle V, E \rangle$ 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.

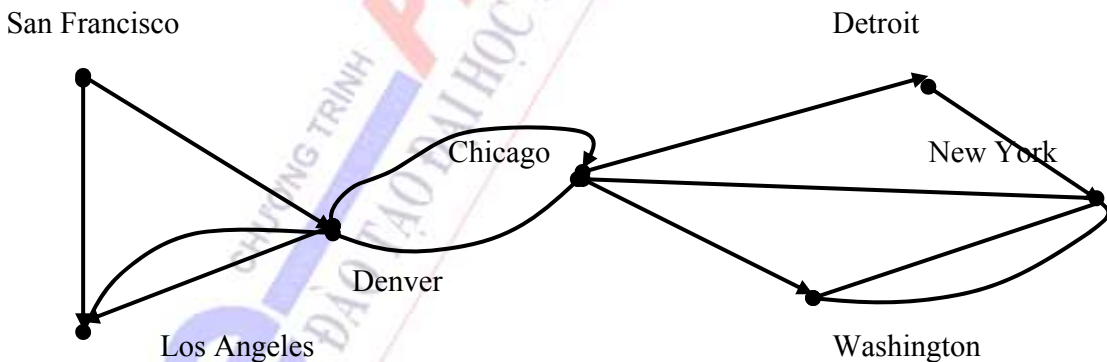
Rõ ràng, mọi đơn đồ thị đều là đa đồ thị, nhưng không phải đa đồ thị nào cũng là đơn đồ thị vì giữa hai đỉnh có thể có nhiều hơn một cạnh nối giữa chúng với nhau. Trong nhiều trường hợp, có máy tính có thể nối nhiều kênh thoại với chính nó. Với loại mạng này, ta không thể dùng đa đồ thị để biểu diễn mà phải dùng giả đồ thị vô hướng. Giả đồ thị vô hướng được mô tả như trong hình 5.3.

Định nghĩa 3. Giả đồ thị vô hướng $G = \langle V, E \rangle$ 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 5.3. Mạng máy tính đa kênh thoại có khuyên.

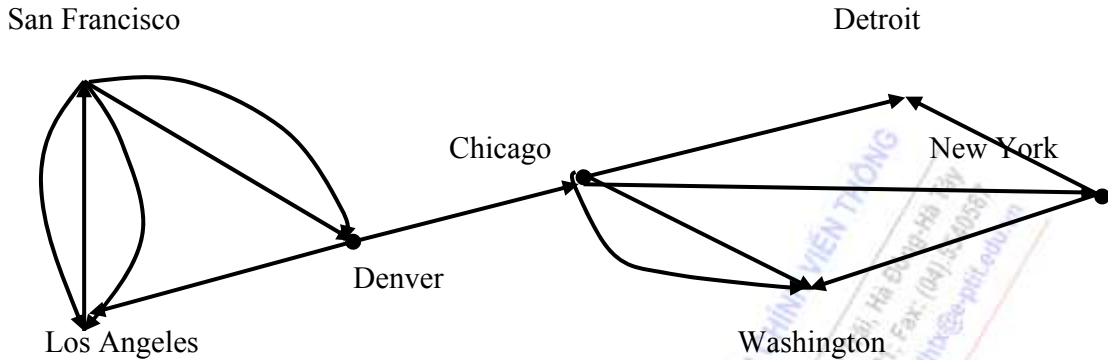
Trong nhiều mạng, các kênh thoại nối giữa hai máy tính có thể chỉ được phép truyền tin theo một chiều. Chẳng hạn máy tính đặt tại San Francisco được phép truy nhập tới máy tính đặt tại Los Angeles, nhưng máy tính đặt tại Los Angeles không được phép truy nhập ngược lại San Francisco. Hoặc máy tính đặt 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 5.4.



Hình 5.4. Mạng máy tính có hướng.

Định nghĩa 4. Đơn đồ thị có hướng $G = \langle V, E \rangle$ 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 5.4 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 5.5.



Hình 5.5. Mạng máy tính đa kênh thoại một chiều.

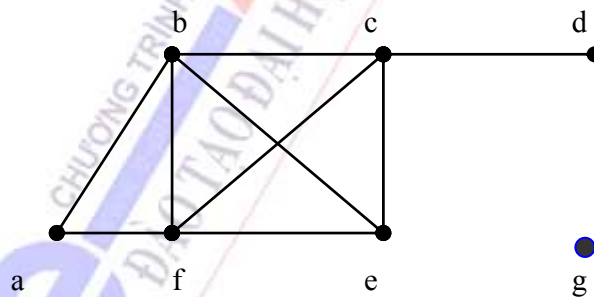
Định nghĩa 5. Đa đồ thị có hướng $G = \langle V, E \rangle$ bao gồm V là tập đỉnh, E là cặp có thứ tự gồm hai phần tử của V được gọi là các cung. Hai cung e_1, e_2 tương ứng với cùng một cặp đỉnh được gọi là cung lặp.

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.

5.1.2. Một số thuật ngữ cơ bản của đồ thị

Định nghĩa 1. Hai đỉnh u và v của đồ thị vô hướng $G = \langle V, E \rangle$ 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 $\text{deg}(v)$.



Hình 5.6 Đồ thị vô hướng G .

Ví dụ 1. Xét đồ thị trong hình 5.6, ta có

$$\text{deg}(a) = 2, \text{deg}(b) = \text{deg}(c) = \text{deg}(f) = 4, \text{deg}(e) = 3, \text{deg}(d) = 1, \text{deg}(g) = 0.$$

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 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) .

5.1.3. 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 $G=\langle V,E \rangle$ là dãy $x_0, x_1, \dots, x_{n-1}, x_n$ trong đó n là số nguyên dương, $x_0=u, x_n=v, (x_i, x_{i+1}) \in E, i=0, 1, 2, \dots, n-1$.

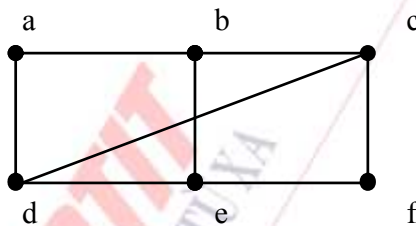
Đường đi như trên còn có thể biểu diễn thành dãy các cạnh

$$(x_0, x_1), (x_1, x_2), \dots, (x_{n-1}, x_n).$$

Ta gọi 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.

Ví dụ 3. Tìm các đường đi, chu trình trong đồ thị vô hướng như trong hình 5.7.

Đễ dàng nhận thấy (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.



Hình 5.7. Đường đi trên đồ thị.

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 3. Đồ thị vô hướng (có hướng) được gọi là liên thông nếu luôn tìm được đường đi giữa hai đỉnh bất kỳ của nó.

5.2. BIỂU DIỄN ĐỒ THỊ TRÊN MÁY TÍNH

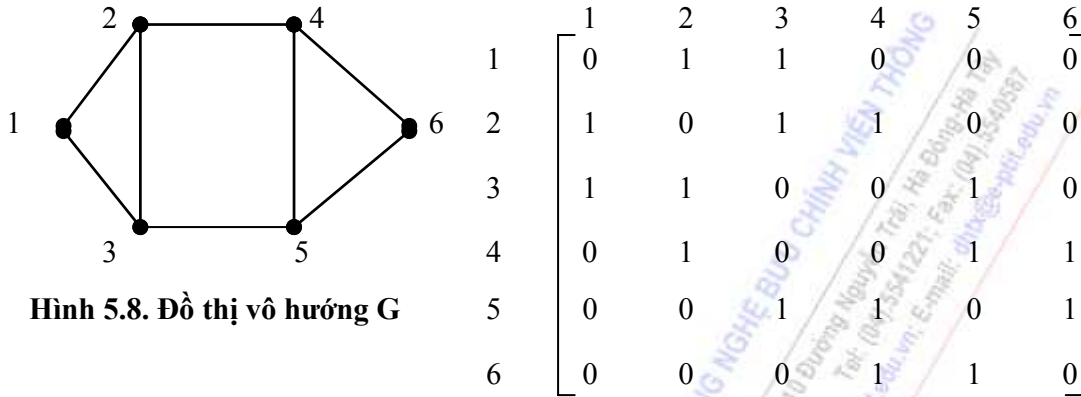
5.2.1. Ma trận kề, ma trận trọng số

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 đơn đồ thị vô hướng $G = \langle V, E \rangle$, với tập đỉnh $V = \{1, 2, \dots, n\}$, tập cạnh $E = \{e_1, e_2, \dots, e_m\}$. 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:

$$A = \{ a_{ij}: a_{ij} = 1 \text{ nếu } (i, j) \in E, a_{ij} = 0 \text{ nếu } (i, j) \notin E; i, j = 1, 2, \dots, n \}.$$

Ví dụ 1. Biểu diễn đồ thị trong hình 5.8 dưới đây bằng ma trận kề.



Hình 5.8. Đồ thị vô hướng G

Ma trận kề có những tính chất sau:

- Ma trận kề của đồ thị vô hướng là ma trận đối xứng $A[i, j] = A[j, i]; i, j = 1, 2, \dots, n$. Ngược lại, mỗi $(0, 1)$ ma trận cấp n đẳng cấu với một đơn đồ thị vô hướng n đỉnh;
- Tổng các phần tử theo dòng i (cột j) của ma trận kề chính bằng bậc đỉnh i (đỉnh j);
- Nếu ký hiệu

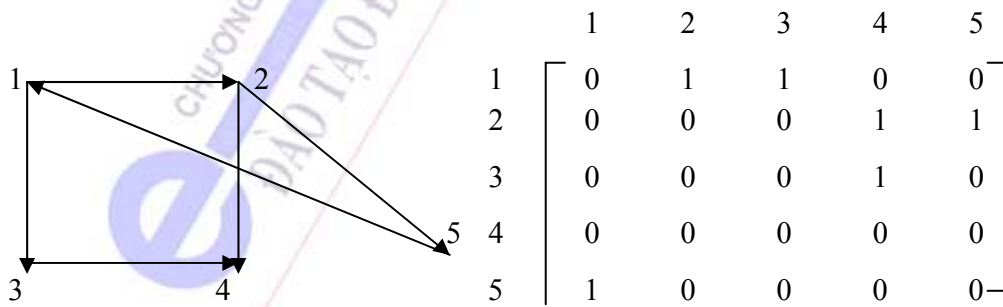
$$a_{ij}^p, i, j = 1, 2, \dots, n \text{ là các phần tử của ma trận}$$

$$A^p = A.A \dots A \text{ (} p \text{ lần) khi đó } a_{ij}^p, i, j = 1, 2, \dots, n,$$

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.

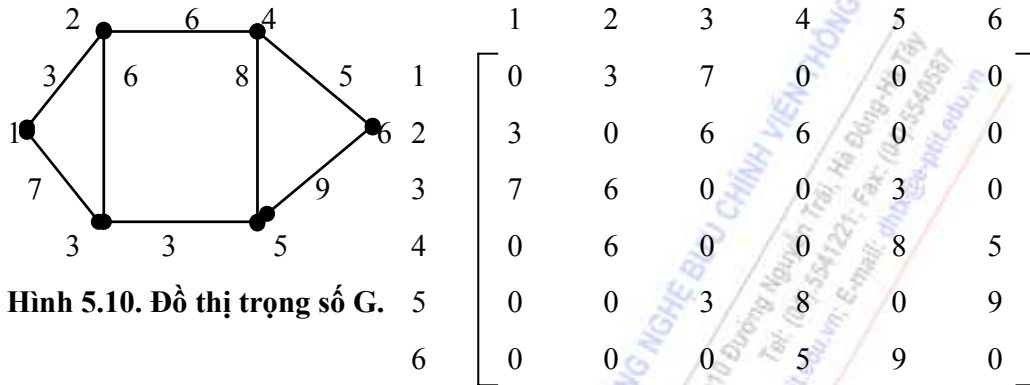
Ví dụ 2. Tìm ma trận kề của đồ thị có hướng trong hình 5.9.



Hình 5.9. 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 $c(e) = d(u, v)$ 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 $c = \{ c[i, j], i, j = 1, 2, \dots, n. c[i, j] = d(i, j) \text{ nếu } (i, j) \in E, c[i, j] = \theta \text{ nếu } (i, j) \notin E$. Trong đó, θ nhận các giá trị: $0, \infty, -\infty$ tùy theo từng tình huống cụ thể của thuật toán.

Ví dụ 3. Ma trận kề của đồ thị có trọng số trong hình 5.10.



Hình 5.10. 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.

5.2.2. 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, đồ thị có hướng, đồ thị trọng số:

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	6	5
5	6			5	6	9

Danh sách cạnh cung hình

Đồ thị có hướng

Danh sách trọng số

5.2.3. 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 5.8, đồ thị có hướng trong hình 5.9 được biểu diễn bằng danh sách kề như sau:

		List(i)			List(i)
Đỉnh	1	2 3		Đỉnh	3 2
	2	1 3 4			4 5
	3	1 2 5			4
	4	2 5 6			5
	5	3 4 6			1
	6	4 5			

5.3. CÁC THUẬT TOÁN TÌM KIẾM TRÊN ĐỒ THỊ

5.3.1 Thuật toán tìm kiếm theo chiều sâu

Rất nhiều thuật toán trên đồ thị được xây dựng dựa 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 duyệt theo chiều sâu (Depth First Search) và duyệt 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*

```
void DFS(int v){
```

```

Thăm_Đỉnh(v); chuaxet[v] = FALSE;
for u ∈ ke(v) {
  if (chuaxet[u])
    DFS( v);
}
}

```

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 :

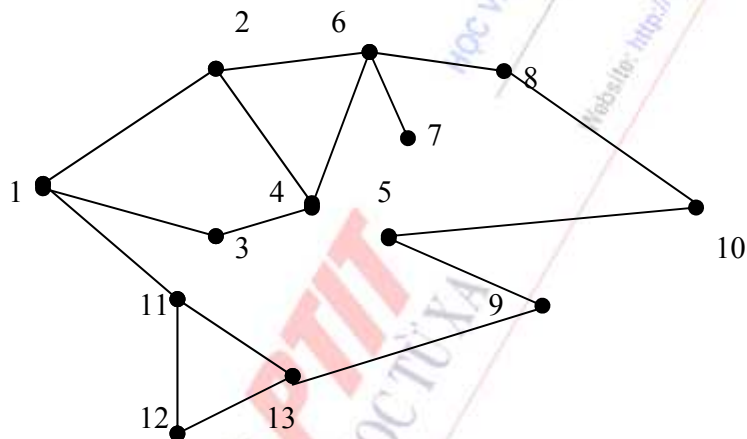
```

for( i=1; i≤n; i++)
  chuaxet[i] = TRUE;
for( i=1; i≤ n; i++)
  if (chuaxet[i])
    DFS( i);

```

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 5.11. Đồ thị vô hướng G

Kết quả duyệt: 1, 2, 4, 3, 6, 7, 8, 10, 5, 9, 13, 11, 12

5.3.2. 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 quy. 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* là v_1, v_2, \dots, v_k được nạp vào queue kế tiếp. Quá trình được thực 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 như dưới đây.

chuaxet- mảng kiểm tra các đỉnh đã xét hay chưa;

queue – hàng đợi lưu trữ các đỉnh sẽ được duyệt của đồ thị;

```
void BFS(int u){
```

```
    queue =  $\phi$ ;
```

```
    u <= queue; (*nạp u vào hàng đợi*)
```

```
    chuaxet[u] = false;
```

```
    while (queue  $\neq \phi$ ){
```

```
        queue <= p; (* lấy p ra từ stack*)
```

```
        Thăm_Đỉnh(p);
```

```
        for v  $\in$  ke(p) {
```

```
            if (chuaxet[v] ) {
```

```
                v <= queue; (*nạp v vào hàng đợi*)
```

```
                chuaxet[v] = false;
```

```
            }
```

```
        }
```

```
}
```

Thuật *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()*.

```
for(u=1; u<=n; u++)
```

```
    chuaxet[u] = TRUE;
```

```
for(u=1; u<=n; u++)
```

```
    if (chuaxet[u])
```

```
        BFS(u);
```

Ví dụ. Áp dụng thuật toán tìm kiếm theo chiều rộng với đồ thị trong hình 5.11 ta nhận được kết quả như sau:

1, 2, 3, 11, 4, 6, 12, 13, 7, 8, 9, 10, 5;

5.3.3. Kiểm tra tính liên thông của đồ thị

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 một 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*.

```
void BFS(int u){
```

```

queue=  $\phi$ ;
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  $\neq \phi$ ) {
    queue<=p; (* lấy p ra từ stack*)
    Thăm_Đỉnh(p);
    for v  $\in$  ke(p) {
        if (chuaxet[v]){
            v<= queue; (* nạp v vào hàng đợi*)
            chuaxet[v] = solt;
        }
    }
}
}
}

```

5.3.4. Tìm đường đi giữa hai đỉnh bất kỳ của đồ thị

Thu 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. Dưới đây là toàn văn chương trình tìm đường đi giữa hai đỉnh của đồ thị.

```

#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;
/* Breadth First Search */
void Init(void){
    FILE *fp; int i, j;
    fp=fopen("lienth.IN", "r");
    if(fp==NULL){
        printf("\n Khong co file input");
        delay(2000);return;
    }
    fscanf(fp,"%d", &n);
    printf("\n So dinh do thi:%d",n);
}

```

```

printf("\n Ma tran ke cua do thi:");

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 Khong co duong di tu %d den %d",s,t);
        getch();
        return;
    }
    printf("\n Duong di tu %d den %d la:",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 Dinh dau:"); scanf("%d",&s);
    printf("\n Dinh cuoi:"); scanf("%d",&t);
    Init();printf("\n");BFS(s);
    n();getch();
    Result();getch();
}

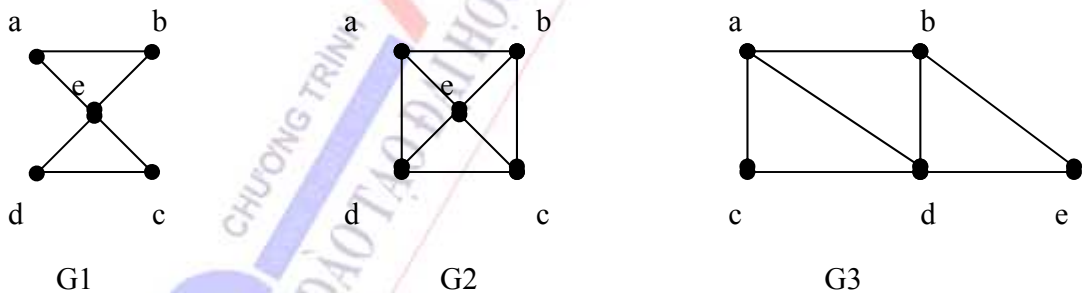
```

5.4. 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 5.12.



Hình 5.12. 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.

Định lý. Đồ thị vô hướng liên thông $G = \langle V, E \rangle$ là đồ thị Euler khi và chỉ khi mọi đỉnh của G đều có bậc chẵn. Đồ thị vô hướng liên thông $G = \langle V, E \rangle$ là đồ thị nửa Euler khi và chỉ khi nó không có quá hai đỉnh bậc lẻ.

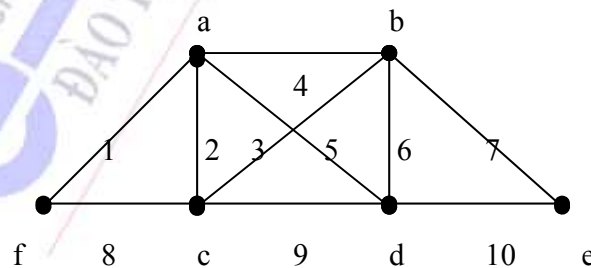
Để tìm một chu trình Euler, ta thực hiện theo thuật toán sau:

- Tạo một mảng CE để ghi đường đi và một stack để xếp các đỉnh ta sẽ xét. Xếp vào đó một đỉnh tùy 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 xóa 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.

```
void Euler_Cycle(int u){
    Stack=φ; CE=φ;
    u=>Stack; { nạp u vào stack};
    while (Stack≠φ) {
        x= top(Stack); { x là phần tử đầu stack }
        if (ke(x) ≠ φ) {
            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};
        }
        else {
            x<= Stack; { lấy x ra khỏi stack};
            CE <=x; { nạp x vào CE};
        }
    }
}
```

Ví dụ. Tìm chu trình Euler trong hình 5.13.



Hình 5.13. Đồ thị vô hướng G .

Các bước thực hiện theo thuật toán sẽ cho ta kết quả sau:

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	

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.

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.

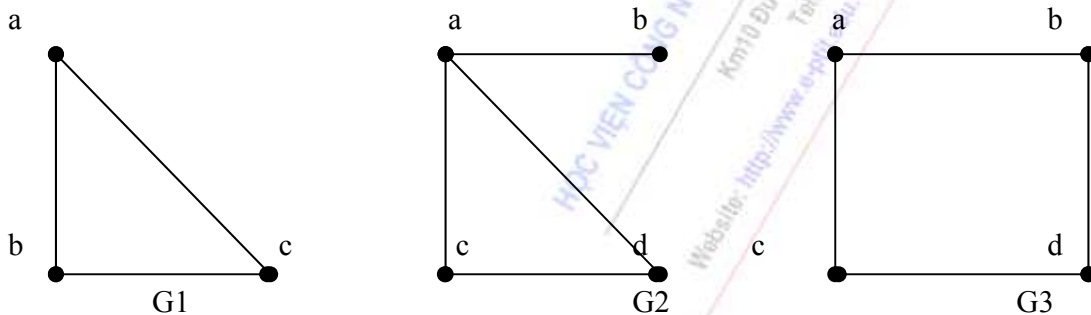
5.5. ĐƯỜNG ĐI VÀ CHU TRÌNH HAMILTON

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 đồ thị hamilton G3, nửa Hamilton G2 và G1.



Hình 5.14. Đồ thị 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 Hamilton của đồ thị, chúng ta có thể sử dụng thuật toán sau:

```
void Hamilton(int k){
    (* Liệt kê các chu trình Hamilton của đồ thị bằng cách phát triển dãy đỉnh
    (X[1], X[2], ..., X[k-1]) của đồ thị G = (V, E) *)
    for y ∈ Ke(X[k-1]) {
        if ((k==n+1) && (y == v0))
            Ghinhan(X[1], X[2], ..., X[n], v0);
        else {
            X[k]=y; chuaxet[y] = false;
            Hamilton(k+1);
            chuaxet[y] = true;
        }
    }
}
```

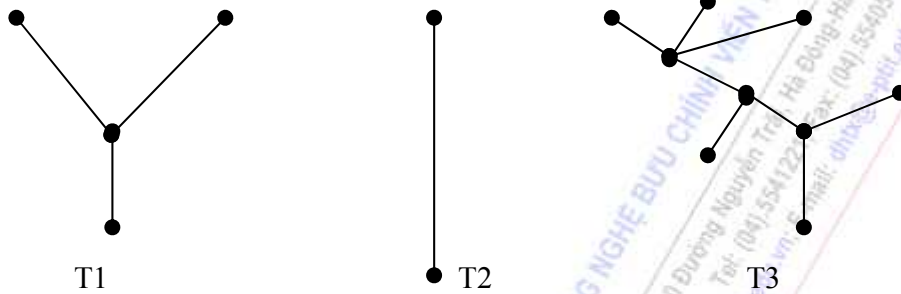
5.6. CÂY BAO TRÙM

5.6.1. Khái niệm và định nghĩa

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 5.15.



Hình 5.15 . 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 $G = \langle V, E \rangle$ là đồ thị vô hướng n đỉnh. Khi đó những khẳng định sau là tương đương

- G là một cây.
- G là đồ thị vô hướng liên thông không có chu trình.
- G liên thông và có đúng $n-1$ cạnh.
- Giữa hai đỉnh bất kỳ của G có đúng một đường đi.
- G liên thông và mỗi cạnh của nó đều là cầu.
- G không chứa chu trình nhưng nếu 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 hay cây khung nếu T thỏa mãn hai điều kiện:

- T là một cây;
- 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 $G = \langle V, E \rangle$ là đồ thị vô hướng liên thông. Hãy xây dựng một cây bao trùm của G .

Bài toán 2. Cho $G = \langle V, E \rangle$ là đồ thị vô hướng liên thông có trọng số. Hãy tìm cây bao trùm nhỏ nhất của G .

5.6.2. 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:

```

void STREE_DFS( int u){
    /* 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) {
        if (chuaxet[v]){
            T = T ∪ (u,v);
            STREE_DFS(v);
        }
    }
}

void STREE_BFS(int u) {
    QUEUE=φ;
    QUEUE<= u; /* đưa u vào hàng đợi*/
    chuaxet[u] = false;
    while (QUEUE ≠ φ) {
        v<= QUEUE; /* lấy v khỏi hàng đợi */
        for p ∈ Ke(v) {
            if (chuaxet[u]) {
                QUEUE<= u;
                chuaxet[u] = false;
                T = T ∪ (v, p);
            }
        }
    }
}

/* Main program */
for u ∈ V {
    chuaxet[u] = true;
    T = φ;
    STREE_BFS(root);
}

```

5.6.3. 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:

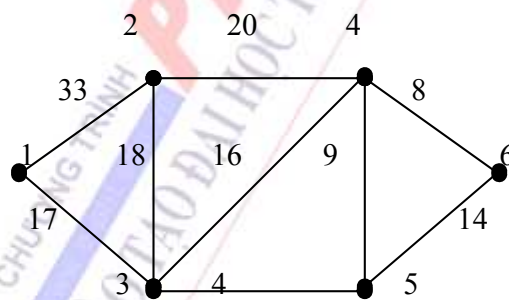
Cho $G = \langle V, E \rangle$ là đồ thị vô hướng liên thông với tập đỉnh $V = \{1, 2, \dots, n\}$ và tập cạnh E gồm m cạnh. Mỗi cạnh e của đồ thị được gán với một số không âm $c(e)$ được gọi là độ dài của nó. Giả sử $H = \langle V, T \rangle$ là một cây bao trùm của đồ thị G . Ta gọi độ dài $c(H)$ của 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 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 \emptyset . 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 bao trùm nhỏ nhất của đồ thị trong hình 5.16.



Hình 5.16. Đồ thị vô hướng liên thông $G = (V, E)$

- Bước 1. Đặt $T = \emptyset$. 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.

5.6.4. Thuật toán Kruskal

Thuật toán xây dựng tập cạnh T của cây khung nhỏ nhất $H = \langle V, T \rangle$ theo từng bước như sau:

- Sắp xếp các cạnh của đồ thị G theo thứ tự tăng dần của trọng số cạnh;
- Xuất phát từ tập cạnh $T = \emptyset$, 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;
- 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:

```
void Kruskal(void) {
    T =  $\emptyset$ ;
    While( | T | < (n-1) and (E  $\neq$   $\emptyset$  ) ) {
        Chọn cạnh e  $\in$  E là cạnh có độ dài nhỏ nhất;
        E = E \ {e};
        if ( T  $\cup$  {e} không tạo nên chu trình )
            T = T  $\cup$  {e};
    }
    if ( | T | < (n-1)
        <Đồ thị không liên thông>;
}
```

5.6.5. 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 tùy 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:

```

void Prim(void) {
    (*bước khởi tạo*)
    Chọn s là một đỉnh nào đó của đồ thị;
     $V_H = \{ s \}$ ;  $T = \phi$ ;  $d[s] = 0$ ;  $near[s] = s$ ;
    For  $v \in V \setminus V_H$  {
         $D[v] = C[s, v]$ ;  $near[v] = s$ ;
    }
    (* Bước lặp *)
    Stop = False;
    While (! stop) {
        Tìm  $u \in V \setminus V_H$  thoả mãn :  $d[u] = \min \{ d[v] \text{ với } u \in V \setminus V_H \}$ ;
         $V_H = V_H \cup \{ u \}$ ;  $T = T \cup \{ u, near[u] \}$ ;
        If ( $| V_H | == n$ ) {
             $H = (V_H, T)$  là cây khung nhỏ nhất của đồ thị;
            Stop := TRUE;
        }
        Else
            For ( $v \in V \setminus V_H$ ) {
                If ( $d[v] > C[u, v]$ ) {
                     $D[v] = C[u, v]$ ;
                     $Near[v] = u$ ;
                }
            }
        }
    }
}

```

5.7. BÀI TOÁN TÌM ĐƯỜNG ĐI NGẮN NHẤT

5.7.1. Phát biểu bài toán

Xét đồ thị có hướng $G = \langle V, E \rangle$; trong đó $| V | = n$, $| E | = m$. Với mỗi cung $(u, v) \in E$, ta đặt tương ứng với nó một số thực $A(u, v)$ được gọi là trọng số của cung. Ta sẽ đặt $A[u, v] = \infty$ nếu $(u, v) \notin E$. Nếu dãy v_0, v_1, \dots, v_k là một đường đi trên G thì $\sum_{i=1}^k A[v_{i-1}, v_i]$ 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 $s \in V$ (đỉnh nguồn) đến đỉnh cuối $t \in V$ (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 $d(s, t) = \infty$. 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.

5.7.2. 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 nhãn 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:

```
void Dijkstra(void) {
(*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*)
(* Bước 1: Khởi tạo nhãn tạm thời cho các đỉnh*)
    for (v=1; v ≤ n; v++) {
        d[v] = A[s,v];
        truoc[v]=s;
    }
    d[s]=0; T = V \ {s}; (*T là tập đỉnh có nhãn tạm thời*)
    while (T! = ∅) { (* bước lặp *)
        Tìm đỉnh u ∈ T sao cho d[u] = min { d[z] : z ∈ T }
        T = T \ {u}; (*cố định nhãn đỉnh u*);
        For (v ∈ T) { (* Gán lại nhãn cho các đỉnh trong T*)
            If ( d[v] > d[u] + A[u, v] ) {
                d[v] = d[u] + A[u, v];
                truoc[v] = u;
            }
        }
    }
}
```

5.7.3. 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 (chỉ ít là $O(n^3)$). Trong trường hợp tổng quát, người ta thường dùng thuật toán *Floy* được mô tả như sau:

```
void Floy(){
(* Tìm đường đi ngắn nhất giữa tất cả các cặp đỉnh
```

Input: Đồ thị cho bởi ma trận trọng số $a[i, j]$, $i, j = 1, 2, \dots, n$.

Output:- Ma trận đường đi ngắn nhất giữa các cặp đỉnh $d[i, j]$, $i, j = 1, 2, \dots, n$;

$d[i, j]$ là độ dài ngắn nhất từ i đến j .

Ma trận ghi nhận đường đi $p[i, j]$, $i, j = 1, 2, \dots, n$

$p[i, j]$ ghi nhận đỉnh đi trước đỉnh j trong đường đi ngắn nhất;*)

(*bước khởi tạo*)

```
for( i=1; i<=; i++)
  for( j=1; j<=n; j++) {
    d[i,j] = a[i, j];
    p[i,j] = i;
  }
(*bước lặp *)
for( k=1; k<=n; k++)
  for( i=1; i<=n; i++)
    for( j=1; j<=n; j++)
      if (d[i,j] > d[i, k] + d[k, j]) {
        d[i, j] = d[i, k] + d[k, j];
        p[i,j] = p[k, j];
      }
}
```

Bạn đọc có thể tìm thấy những cài đặt cụ thể các thuật toán trên đồ thị thông qua các tài liệu [1], [5].



HỌC VIỆN CÔNG NGHỆ BƯU CHÍNH VIỄN THÔNG
Km10 Đường Nguyễn Trãi, Hà Đông-Hà Tây
Tel: (04) 5541221; Fax: (04) 5540587
Website: <http://www.c-ptit.edu.vn>; E-mail: dhk@ptit.edu.vn

NHỮNG NỘI DUNG CẦN GHI NHỚ

- ✓ Nắm vững những khái niệm và định nghĩa cơ bản của đồ thị.
- ✓ Hiểu được các phương pháp biểu diễn đồ thị trên máy tính.
- ✓ Nắm vững được các thuật toán tìm kiếm trên đồ thị: thuật toán tìm kiếm theo chiều rộng, thuật toán tìm kiếm theo chiều sâu và ứng dụng của nó trong bài toán tìm đường đi giữa hai đỉnh của đồ thị cũng như trong tính toán các thành phần liên thông của đồ thị.
- ✓ Nắm vững sự khác biệt giữa đồ thị Euler và đồ thị Hamilton cùng với thuật toán tìm đường đi và chu trình trên các đồ thị Euler & Hamilton.
- ✓ Nắm vững bài toán tìm cây bao trùm & tìm cây bao trùm nhỏ nhất.
- ✓ Hiểu & cài đặt nhuần nhuyễn các thuật toán tìm đường đi ngắn nhất giữa các cặp đỉnh của đồ thị trọng số.

BÀI TẬP CHƯƠNG 5

Bài 1. Cho trước ma trận kề của đồ thị. Hãy viết chương trình tạo ra danh sách kề của đồ thị đó.

Bài 2. Cho trước danh sách kề của đồ thị, hãy tạo nên ma trận kề của đồ thị.

Bài 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 file king.out.

Bài 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 file ma.out.

Bài 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).

Bài 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 $[A, A_1, n_1], [A_1, A_2, n_2] \dots [A_k, B, n_k]$ với A_i là điểm đến của tuyến i nhưng lại là điểm khởi

hành của tuyến $i+1$, n_i 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 ($n_1+n_2+\dots+n_k$).

Cho file dữ liệu kiểu text hanhtrinh.in được ghi theo từng dòng, số các dòng trong file 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.

Bài 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 tám 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	Hình 2	Hình 3	Hình 4																																
<table border="1" style="border-collapse: collapse; width: 50px; height: 50px;"> <tr><td>1</td><td>2</td><td>3</td><td>4</td></tr> <tr><td>8</td><td>7</td><td>6</td><td>5</td></tr> </table>	1	2	3	4	8	7	6	5	<table border="1" style="border-collapse: collapse; width: 50px; height: 50px;"> <tr><td>8</td><td>7</td><td>6</td><td>5</td></tr> <tr><td>1</td><td>2</td><td>3</td><td>4</td></tr> </table>	8	7	6	5	1	2	3	4	<table border="1" style="border-collapse: collapse; width: 50px; height: 50px;"> <tr><td>4</td><td>1</td><td>2</td><td>3</td></tr> <tr><td>5</td><td>8</td><td>7</td><td>6</td></tr> </table>	4	1	2	3	5	8	7	6	<table border="1" style="border-collapse: collapse; width: 50px; height: 50px;"> <tr><td>1</td><td>7</td><td>2</td><td>4</td></tr> <tr><td>8</td><td>6</td><td>3</td><td>5</td></tr> </table>	1	7	2	4	8	6	3	5
1	2	3	4																																
8	7	6	5																																
8	7	6	5																																
1	2	3	4																																
4	1	2	3																																
5	8	7	6																																
1	7	2	4																																
8	6	3	5																																

Cho file 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 file 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

Bài 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 $A[i,j] \geq 0$, $i \neq j$. Nếu đường truyền tin từ nút i_1 đến nút i_k phải thông qua các nút i_2, \dots, i_{k-1} thì chi phí truyền thông được tính bằng tổng các chi phí truyền thông $A[i_1, i_2], A[i_2, i_3], \dots, A[i_{k-1}, i_k]$. 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 file TEXT có tên INP.NN. Trong đó, dòng thứ nhất ghi ba số N, i, j , dòng thứ $k + 1$ ghi $k-1$ số $A[k,1], A[k,2], \dots, A[k,k-1]$, $1 \leq k \leq N$.

Kết quả thông báo ra file 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 .

Bài 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 $A[i,j] \geq 0$, $i \neq j$. Nếu đường truyền tin từ nút i_1 đến nút i_k phải thông qua các nút i_2, \dots, i_{k-1} thì chi phí truyền thông được tính bằng tổng các chi phí truyền thông $A[i_1, i_2], A[i_2, i_3], \dots, A[i_{k-1}, i_k]$. 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 file TEXT có tên INP.NN. Trong đó, dòng thứ nhất ghi số N , dòng thứ $k + 1$ ghi $k-1$ số $A[k,1], A[k,2], \dots, A[k,k-1]$, $1 \leq k \leq N$.

Kết quả thông báo ra file 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.

Bài 10. Cho file dữ liệu được tổ chức giống như bài 6.6. Hãy tìm tất cả các hành trình đi từ điểm s đến t .

Bài 11. Cho file 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.

Bài 12. Cho file 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.

Bài 13. Tìm hiểu thuật toán leo đồi trên đồ thị và ứng dụng của nó trong lĩnh vực trí tuệ nhân tạo.

CHƯƠNG 6: SẮP XẾP VÀ TÌM KIẾM (SORTING AND SEARCHING)

Sắp xếp & tìm kiếm là bài toán cơ bản nhất của tin học. Có thể nói, mọi tương tác giữa con người và hệ thống máy tính về bản chất đều là tìm kiếm và thu thập thông tin. Ấn sau các quá trình tìm kiếm là việc sắp xếp các đối tượng theo một trật tự nào đó để quá trình tìm kiếm diễn ra nhanh nhất, chính xác và hiệu quả nhất đó là ý nghĩa cơ bản của quá trình sắp xếp. Nội dung chính của chương này tập chung vào các giải thuật sắp xếp và tìm kiếm cơ bản dưới đây:

- ✓ Giải thuật Selection Sort, Giải thuật Insert Sort, Giải thuật Bubble Sort, Giải thuật Shaker Sort, Giải thuật Quick Sort, Giải thuật Heap Sort, và giải thuật Merge Sort.
- ✓ Tìm kiếm tuần tự (Sequential), tìm kiếm nhị phân (Binary Search) & tìm kiếm trên cây nhị phân (Binary Search).

Bạn đọc có thể tìm thấy những cài đặt cụ thể và những kiến thức sâu hơn trong tài liệu [1] & [6].

6.1. 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 (Process Control Problem), bài toán lập lịch cho CPU (CPU Scheduling), 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

6.2. 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, \dots, k_n với $i = 0, 1, \dots, n$; $k_i < k_{i+1} < \dots, k_n$ và đổi chỗ cho phần tử thứ k_i . Như vậy, sau $j = n-1$ lần chọn, chúng ta sẽ có 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.
-
- 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 .
- 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	k_i	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 Tao lap day so:");
    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 Nhap n="); scanf("%d",&n);
        A=(int *) malloc(n*sizeof(int));
        Init(A,n);Select(A,n);
        free(A);
    }

```

6.3. GIẢI THUẬT INSERTION SORT

Giải thuật Insert Sort được thực hiện dựa trên kinh nghiệm của những người chơi bài. Khi 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 $i-1, i-2, \dots$ 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 tiếp 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.
-
- 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 Tao lap day so:");
    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 Nhap n="); scanf("%d",&n);
    A=(int *) malloc(n*sizeof(int));
    Init(A,n);Insert(A,n);
    free(A);
}
```

6.4. 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 $n-1, n-2, \dots, n-k+1$. 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 Tao lap day so:");
    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 Ket qua lan:%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 Nhap n="); scanf("%d",&n);
    A=(int *) malloc(n*sizeof(int));
    Init(A,n);Bubble(A,n);
    free(A);
}

```

6.5. 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 Tao lap day so:");
    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 Ket qua lan:");
        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 Nhap n="); scanf("%d",&n);
    A=(int *) malloc(n*sizeof(int));
    Init(A,n);Shaker(A,n);
    free(A);
}
```

VIỆN CÔNG NGHỆ BƯU CHÍNH VIỄN THÔNG
Km10 Đường Nguyễn Trãi, Hà Đông-Hà Tây
Tel: (04) 5541221; Fax: (04) 5540587
Website: <http://www.ct-ptit.edu.vn>; E-mail: dhk@ptit.edu.vn

CHƯƠNG TRÌNH PTIT
ĐÀO TẠO ĐẠI HỌC TỪ XA

6.6. 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 trao 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 $C_{\max} = C_{\text{tb}} = O(n \log_2 n)$

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 Tao lap day so:");
    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]);
}
```

HỌC VIỆN CÔNG NGHỆ BƯU CHÍNH VIỄN THÔNG

Km10 Đường Nguyễn Trãi, Hà Đông-Hà Tây
Tel: (04) 5541 221; Fax: (04) 5540 587

Website: <http://www.e-ptit.edu.vn>; E-mail: dhk@e-ptit.edu.vn

```

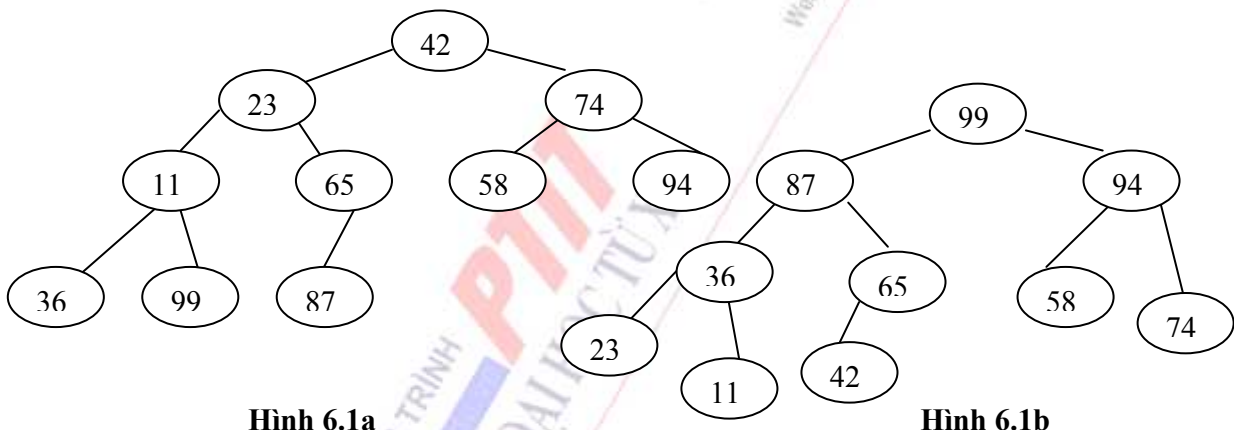
        delay(1000);
    }
    void main(void){
        int *A,n;clrscr();
        printf("\n Nhap n="); scanf("%d",&n);
        A=(int *)malloc(n*sizeof(int));
        Init(A,n);Quick(A,n);printf("\n");
        In(A,n);getch();
        free(A);
    }

```

6.7. 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 khóa 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 bằng khóa 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 6.1a thì heap của nó là 6.1b.



Hình 6.1a

Hình 6.1b

Để chuyển cây nhị phân 6.1a thành cây nhị phân 6.1b 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 khóa 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 Tao lap day so:");
    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;
```

HỌC VIỆN CÔNG NGHỆ BƯU CHÍNH VIỄN THÔNG
Km10 Đường Nguyễn Trãi, Hà Đông-Hà Tây
Tel: (04) 5541221; Fax: (04) 5540587
Website: <http://www.c-ptit.edu.vn>; E-mail: dhk@ptit.edu.vn

CHƯƠNG TRÌNH PTIT
ĐẠO TẠO HỌC TỬ XA

```

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 Nhap n="); scanf("%d",&n);
    A=(int *) malloc(n*sizeof(int));
    Init(A,n);Heap(A,n);printf("\n");
    In(A,n);getch();
    free(A);
}

```

6.8. 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 như 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.

-
- 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 Tao lap day so:");
    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 Ket qua:");
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 Nhap n="); scanf("%d",&n);
    A=(int *) malloc(n*sizeof(int));
    Init(A,n);Merge(A,n);printf("\n");
    free(A);
}

```

6.9. 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, \dots, 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 xung thêm bản ghi mới có giá trị khóa là X thì giải thuật được gọi là giải thuật tìm kiếm bổ sung.

6.9.1. 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 Tao lap day so:");
    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 Nhap 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();
}

```

6.9.2. 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, high = n-1;
    while (low<=high){
        mid = (low +high)/2;
        if (X >A[mid] ) low = mid +1;
        else if (X<A[mid] ) high = mid -1;
        else return (mid);
    }
    return(-1);
}

void Init(int *A, int n){
    int i;
    printf("\n Tao lap day so:");
    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 Ket qua lan:%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();
```

HỌC VIỆN CÔNG NGHỆ BƯU CHÍNH VIỄN THÔNG
Km10 Đường Nguyễn Trãi, Hà Đông-Hà Tây
Tel: (04) 5541221; Fax: (04) 5540587
Website: <http://www.ct-ptit.edu.vn>; E-mail: dhk@ptit.edu.vn

PTIT
ĐƯỜNG TRẦN
ĐẠO TẠO ĐẠI HỌC TÀI

```
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);
}
```

HỌC VIỆN CÔNG NGHỆ BƯU CHÍNH VIỄN THÔNG

Km10 Đường Nguyễn Trãi, Hà Đông-Hà Tây
Tel: (04) 5541221; Fax: (04) 5540587
Website: <http://www.ct-ptit.edu.vn>; E-mail: dhk@ptit.edu.vn

CHƯƠNG TRÌNH **PTIT**
ĐÀO TẠO ĐẠI HỌC TỪ XA

NHỮNG NỘI DUNG CẦN GHI NHỚ

- ✓ Hiểu được ý nghĩa vai trò của bài toán sắp xếp và tìm kiếm trong tin học.
- ✓ Cài đặt nhuần nhuyễn các giải thuật sắp xếp và tìm kiếm trên các cấu trúc dữ liệu khác nhau.
- ✓ Giải quyết các bài tập thực hành kèm theo làm thăng tiến kỹ năng giải quyết bài toán sắp xếp & tìm kiếm.



BÀI TẬP CHƯƠNG 6

- Bài 1.** Cài đặt chương trình theo thuật toán Quick Sort không dùng phương pháp đệ qui mà dùng cấu trúc stack.
- Bài 2.** Tìm hiểu về giải thuật Shell-Sort là phương pháp cải tiến của Insertion Sort.
- Bài 3.** Cài đặt lại giải thuật Bubble Sort sao cho các node nhỏ được đẩy dần về phía trước.
- Bài 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.
- Bài 5.** Cài đặt giải thuật Bubble Sort trên file.
- Bài 6.** Cài đặt giải thuật Insertion Sort trên file.
- Bài 7.** Cài đặt giải thuật Quick Sort trên file.
- Bài 8.** Cài đặt các giải thuật sắp xếp theo nhiều khoá khác nhau.
- Bài 9.** Nghiên cứu và cài đặt thuật toán tìm kiếm tam phân.
- Bài 10.** Nghiên cứu và cài đặt thuật toán sắp xếp kiểu hoà nhập thực hiện trên file.
- Bài 11.** Viết chương trình chuyển đổi một file dữ liệu được tổ chức theo khuôn dạng *.DBF thành file kiểu text. Ngược lại, chuyển đổi file dữ liệu kiểu text thành một file dữ liệu theo khuôn dạng DBF.
- Bài 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.

TÀI LIỆU THAM KHẢO

- [1] Lê Hữu Lập - Nguyễn Duy Phương. *Giáo trình Kỹ thuật lập trình*. NXB Bưu Điện, 2002.
- [2] Đỗ Xuân Lôi. *Cấu trúc dữ liệu và giải thuật*. NXB Khoa Học Kỹ Thuật, 2000.
- [3] Đặng Huy Ruận. *Lý thuyết đồ thị*. NXB Khoa Học Kỹ Thuật, 2003.
- [4] William Ford, William Topp. *Data Structures with C++*. Prentice Hall, 1996.
- [5] Mark Allen Weiss. *Data Structures and Algorithm Analysis In C*. Prentice Hall, 1996.
- [6] Phan Đăng Cầu. *Cấu trúc dữ liệu và Giải thuật* (Tài liệu giảng dạy–Học Viện Công nghệ BCVT), 2003.



HỌC VIỆN CÔNG NGHỆ BƯU ĐIỆN VÀ TRUYỀN THÔNG
Km10 Đường Nguyễn Trãi, Cầu Giấy, Hà Nội
Tel: 04.554122 Fax: 04.5540550
Website: <http://www.e-ptit.edu.vn> Email: info@e-ptit.edu.vn

MỤC LỤC

Chương 1: ĐẠI CƯƠNG VỀ KỸ THUẬT LẬP TRÌNH CẤU TRÚC	3
1.1. Sơ lược về lịch sử lập trình cấu trúc.....	3
1.2. Cấu trúc lệnh, lệnh có cấu trúc, cấu trúc dữ liệu	5
1.2.1. Cấu trúc lệnh (cấu trúc điều khiển).....	5
1.2.2. Lệnh có cấu trúc.....	7
1.2.3. Cấu trúc dữ liệu.....	7
1.3. Nguyên lý tối thiểu	8
1.3.1. Tập các phép toán	8
1.3.2. Tập các lệnh vào ra cơ bản.....	11
1.3.3. Thao tác trên các kiểu dữ liệu có cấu trúc.....	11
1.4. Nguyên lý địa phương.....	13
1.5. Nguyên lý nhất quán.....	15
1.6. Nguyên lý an toàn.....	16
1.7. Phương pháp Top-Down	18
1.8. Phương pháp Bottom - Up.....	22
Chương 2: DUYỆT VÀ ĐỆ QUI	29
2.1. Định nghĩa bằng đệ qui	29
2.2. Giải thuật đệ qui	30
2.3. Thuật toán sinh kế tiếp	31
2.4. Thuật toán quay lui (Back track).....	34
2.5. Thuật toán nhánh cận	37
Chương 3: NGĂN XẾP, HÀNG ĐỢI VÀ DANH SÁCH MỐC NÓI (STACK, QUEUE, LINK LIST)	51
3.1. Kiểu dữ liệu ngăn xếp và ứng dụng.....	51
3.1.1. Định nghĩa và khai báo	51
3.1.2. Các thao tác với stack	53
3.1.3. Ứng dụng của stack.....	53

3.2.	Hàng đợi (Queue).....	55
3.2.1.	Định nghĩa và khai báo	55
3.2.2.	Ứng dụng hàng đợi.....	57
3.3.	Danh sách liên kết đơn	62
3.3.1.	Giới thiệu và định nghĩa.....	62
3.3.2.	Các thao tác trên danh sách móc nối.....	63
3.4.	Danh sách liên kết kép.....	67
Chương 4: CẤU TRÚC DỮ LIỆU CÂY (TREE)		77
4.1.	Định nghĩa và khái niệm	77
4.2.	Cây nhị phân.....	78
4.3.	Biểu diễn cây nhị phân	81
4.3.1.	Biểu diễn cây nhị phân bằng danh sách tuyến tính.....	81
4.3.2.	Biểu diễn cây nhị phân bằng danh sách móc nối	82
4.4.	Các thao tác trên cây nhị phân.....	83
4.4.1.	Định nghĩa cây nhị phân bằng danh sách tuyến tính.....	83
4.4.2.	Định nghĩa cây nhị phân theo danh sách liên kết.....	83
4.4.3.	Các thao tác trên cây nhị phân	83
4.5.	Các phép duyệt cây nhị phân (Traversing Binary Tree).....	88
4.5.1.	Duyệt theo thứ tự trước (Preorder Traversal).....	88
4.5.2.	Duyệt theo thứ tự giữa (Inorder Traversal)	89
4.5.3.	Duyệt theo thứ tự sau (Postorder Traversal)	89
4.6.	Cài đặt cây nhị phân tìm kiếm.....	90
Chương 5: ĐỒ THỊ (GRAPH)		103
5.1.	Những khái niệm cơ bản của đồ thị.....	103
5.1.1.	Các loại đồ thị	103
5.1.2.	Một số thuật ngữ cơ bản của đồ thị	106
5.1.3.	Đường đi, chu trình, đồ thị liên thông.....	107
5.2.	Biểu diễn đồ thị trên máy tính	107
5.2.1.	Ma trận kề, ma trận trọng số	107
5.2.2.	Danh sách cạnh (cung).....	109
5.2.3.	Danh sách kề.....	110

5.3.	Các thuật toán tìm kiếm trên đồ thị	110
5.3.1.	Thuật toán tìm kiếm theo chiều sâu	110
5.3.2.	Thuật toán tìm kiếm theo chiều rộng (Breadth First Search).....	111
5.3.3.	Kiểm tra tính liên thông của đồ thị.....	112
5.3.4.	Tìm đường đi giữa hai đỉnh bất kỳ của đồ thị	113
5.4.	Đường đi và chu trình Euler	115
5.5.	Đường đi và chu trình Hamilton.....	118
5.6.	Cây bao trùm	119
5.6.1.	Khái niệm và định nghĩa	119
5.6.2.	Tìm một cây bao trùm trên đồ thị.....	120
5.6.3.	Tìm cây bao trùm ngắn nhất.....	121
5.6.4.	Thuật toán Kruskal.....	122
5.6.5.	Thuật toán Prim.....	122
5.7.	Bài toán tìm đường đi ngắn nhất	123
5.7.1.	Phát biểu bài toán.....	123
5.7.2.	Thuật toán Dijkstra.....	124
5.7.3.	Thuật toán Floy	124
Chương 6: SẮP XẾP VÀ TÌM KIẾM (SORTING AND SEARCHING).....		131
6.1.	Đặt bài toán	131
6.2.	Giải thuật Selection Sort.....	132
6.3.	Giải thuật Insertion Sort	134
6.4.	Giải thuật Bubble Sort.....	136
6.5.	Giải thuật Shaker Sort	137
6.6.	Giải thuật Quick Sort.....	139
6.7.	Giải thuật Heap Sort.....	141
6.8.	Giải thuật Merge Sort	143
6.9.	Tìm kiếm (Searching).....	145
6.9.1.	Tìm kiếm tuần tự (Sequential Searching)	146
6.9.2.	Tìm kiếm nhị phân (Binary Searching).....	147
TÀI LIỆU THAM KHẢO		152
MỤC LỤC.....		153