TRƯỜNG ĐẠI HỌC SƯ PHẠM KỸ THUẬT VĨNH LONG
KHOA CÔNG NGHỆ THÔNG TIN

GIÁO TRÌNH

CẤU TRÚC DỮ LIỆU VÀ GIẢI THUẬT

(DATA STRUCTURE AND ALGORITHM)

BIÊN SOẠN
ThS. GVC. NGUYỄN VĂN HIẾU
ThS. LÊ THỊ HOÀNG YẾN
ThS. MAI THIÊN THƯ

Vĩnh Long - 2022

LỜI NÓI ĐẦU

-----ᘓ✧ᘒ-----

Để đáp ứng nhu cầu học tập và nghiên cứu của các bạn sinh viên, đặc biệt là sinh viên chuyên ngành Công nghệ thông tin, Khoa Công nghệ thông tin - Trường Đại học Sư phạm Kỹ thuật Vĩnh Long đã tiến hành biên soạn các giáo trình, bài giảng chính trong chương trình học. Giáo trình môn Cấu trúc dữ liệu và Giải thuật này được biên soạn cơ bản dựa trên quyển "Data Structures and Algorithms" của Alfred V. Aho, John E. Hopcroft và Jeffrey D. Ullman do Addison-Wesley tái bản. Giáo trình này cũng được biên soạn dựa trên kinh nghiệm giảng dạy nhiều năm môn Cấu trúc dữ liệu và Giải thuật của các giáo viên trong khoa chúng tôi. Ngoài ra chúng tôi cũng đã tham khảo rất nhiều tài liệu của các trường đại học trong và ngoài nước.

Tài liệu này được biên soạn dựa theo đề cương chi tiết môn học Cấu trúc dữ liệu và Giải thuật của Khoa Công nghệ thông tin Trường Đại học Sư phạm Kỹ thuật Vĩnh Long dùng cho sinh viên chuyên ngành Công nghệ thông tin bao gồm 6 chương:

Chương 1: Ngôn ngữ lập trình C++

Chương 2: Tổng quan về cấu trúc dữ liệu và giải thuật

Chương 3: Danh sách

Chương 4: Cây

Chương 5: Đồ thị

Chương 6: Bảng băm

Mục tiêu của nó nhằm giúp các bạn sinh viên chuyên ngành có một tài liệu cô đọng dùng làm tài liệu học tập và nghiên cứu nhưng chúng tôi cũng không loại trừ toàn bộ các đối tượng khác tham khảo. Chúng tôi nghĩ rằng các bạn sinh viên thuộc các chuyên ngành khác và những người quan tâm đến cấu trúc dữ liệu và giải thuật sẽ tìm được những điều bổ ích trong quyển giáo trình này.

Mặc dù chúng tôi đã rất cố gắng trong quá trình biên soạn nhưng chắc chắn giáo trình này sẽ còn nhiều thiếu sót và hạn chế. Rất mong nhận được sự đóng góp ý kiến quý báu của các sinh viên và bạn đọc để giáo trình ngày càng hoàn thiện hơn.

Chân thành cảm ơn!

Nhóm tác giả

CHƯƠNG 1: NGÔN NGỮ LẬP TRÌNH C++

-----ᘓ✧ᘒ-----

1. CÁC CẤU TRÚC ĐIỀU KHIỂN

1.1. Cấu trúc rẽ nhánh

1.1.1. Cấu trúc if

Ý nghĩa

Một câu lệnh if cho phép chương trình có thể thực hiện khối lệnh này hay khối lệnh khác phụ thuộc vào điều kiện được viết trong câu lệnh là đúng hay sai. Nói cách khác câu lệnh if cho phép chương trình rẽ nhánh (chỉ thực hiện 1 trong 2 nhánh).

Cú pháp

```
if(điều kiện)
    khối lệnh 1;
[else
    khối lệnh 2;]
```

Trong cú pháp trên câu lệnh if có hai dạng: có else và không có else. Điều kiện là một biểu thức logic tức nó có giá trị đúng (khác 0) hoặc sai (bằng 0).

Khi chương trình thực hiện câu lệnh if nó sẽ tính biểu thức điều kiện. Nếu điều kiện đúng chương trình sẽ tiếp tục thực hiện các lệnh trong khối lệnh 1, ngược lại nếu điều kiện sai chương trình sẽ thực hiện khối lệnh 2 (nếu có else) hoặc không làm gì (nếu không có else).

Đặc điểm

- Đặc điểm chung của các câu lệnh có cấu trúc là bản thân nó chứa các câu lệnh khác. Điều này cho phép các câu lệnh if có thể lồng nhau.

- Nếu nhiều câu lệnh if (có else và không else) lồng nhau việc hiểu if và else nào đi với nhau cần phải chú ý. Qui tắc là else sẽ đi với if gần nó nhất mà chưa được ghép cặp với else khác.

Các lưu ý

- Điều kiện (hay biểu thức) sau if phải đặt trong cặp dấu ngoặc đơn ( ).

- Nếu khối lệnh có nhiều hơn 1 câu lệnh thì phải bao chúng trong cặp dấu ngoặc móc { }.

- Nếu khối lệnh chỉ có 1 câu lệnh thì ta có thể viết ngay sau if hoặc else mà không cần phải xuống hàng.

Chương trình minh họa 1

Viết chương trình nhập vào họ tên, tên môn học và điểm (thang điểm 10) cho một sinh viên, hãy qui về điểm chữ tương ứng, rồi từ đó qui về thang điểm 4 của môn học đó, hãy in họ tên, môn học và cả 3 loại điểm cho sinh viên đó.

```
#include <iostream>
#include <string.h>
using namespace std;
main()
{   char *ht, *mh, *dc;
    float d10, d4;
    cout<<"Nhap ho ten sinh vien:";
    ht=new char[40];
    cin.getline(ht,40);
    cout<<"Nhap ten mon hoc:";
    mh=new char[40];
    cin.getline(mh,40);
    cout<<"Nhap diem so:";
    cin>>d10;
    dc=new char[3];
    if(d10>=8.5) dc="A";
    else if(d10>=7.8) dc="B+";
         else if(d10>=7) dc="B";
              else if(d10>=6.3) dc="C+";
                   else if(d10>=5.5) dc="C";
                        else if(d10>=4.8) dc="D+";
                             else if(d10>=4) dc="D";
                                  else dc="F";
    if(dc=="A") d4=4;
    else if(dc=="B+") d4=3.5;
         else if(dc=="B") d4=3;
              else if(dc=="C+") d4=2.5;
                   else if(dc=="C") d4=2;
                        else if(dc=="D+") d4=1.5;
                             else if(dc=="D") d4=1;
                                  else d4=0;
    cout<<"Sinh vien "<<strupr(ht)<<" co diem mon hoc ";
    cout<<strupr(mh)<<" la:"<<endl;
    cout<<"Thang diem 10: "<<d10<<endl;
```

```
    cout<<"Diem chu: "<<dc<<endl;
    cout<<"Thang diem 4: "<<d4<<endl;
}
```

1.1.2. Cấu trúc switch

Ý nghĩa

Câu lệnh if cho ta khả năng được lựa chọn một trong hai nhánh để thực hiện, do đó nếu muốn rẽ theo nhiều nhánh ta phải sử dụng cấu trúc if lồng nhau. Tuy nhiên trong trường hợp như vậy chương trình sẽ phức tạp và khó đọc hơn, vì vậy C++ cung cấp thêm một câu lệnh cấu trúc khác cho phép chương trình có thể chọn một trong nhiều nhánh để thực hiện, đó là câu lệnh switch.

Cú pháp

```
switch(biểu thức điều khiển)
     {
          case biểu_thức_1: dãy lệnh 1;
          case biểu_thức_2: dãy lệnh 2;
          …
          case biểu_thức_n: dãy lệnh n;
          [default: dãy lệnh n+1;]
     }
```

- Biểu thức điều khiển: phải có kiểu nguyên hoặc ký tự.

- Các biểu_thức_i: được tạo từ các hằng nguyên hoặc ký tự.

- Các dãy lệnh có thể rỗng và không cần bao dãy lệnh bởi cặp dấu ngoặc móc { }.

- Nhánh default có thể có hoặc không và vị trí của nó có thể nằm bất kỳ trong câu lệnh nào (giữa các nhánh case) tuy nhiên thông thường người ta đặt nó nằm ở cuối cùng.

Cách thực hiện

Để thực hiện câu lệnh switch đầu tiên chương trình tính giá trị của biểu thức điều khiển (btđk), sau đó so sánh kết quả của btđk với giá trị của các biểu_thức_i bên dưới lần lượt từ biểu thức đầu tiên (thứ nhất) cho đến biểu thức cuối cùng (thứ n), nếu giá trị của btđk bằng giá trị của biểu thức thứ i đầu tiên nào đó thì chương trình sẽ thực hiện dãy lệnh thứ i và tiếp tục thực hiện tất cả dãy lệnh còn lại (từ dãy lệnh thứ i+1) cho đến hết (gặp dấu ngoặc đóng } của lệnh switch). Nếu quá trình so sánh không gặp biểu thức (nhánh case) nào bằng với giá trị của btđk thì chương trình thực hiện dãy lệnh trong default và tiếp tục cho đến hết (sau default có thể còn những nhánh case khác). Trường hợp câu lệnh switch không có nhánh default và btđk không khớp với bất kỳ nhánh case nào thì chương trình không làm gì, coi như đã thực hiện xong lệnh switch.

Nếu muốn lệnh switch chỉ thực hiện nhánh thứ i (khi btđk == biểu_thức_i) mà không phải thực hiện thêm các lệnh còn lại của các nhánh khác thì cuối dãy lệnh thứ i thông thường ta đặt thêm lệnh break; đây là lệnh cho phép thoát ra khỏi một lệnh cấu trúc bất kỳ.

Chương trình minh họa 2

Viết chương trình mô phỏng một menu cho phép lựa chọn nhiều công việc khác nhau trên danh sách như: Nhập, In, Thêm, Xóa, Tìm, Thoát.

```
#include <iostream>
using namespace std;
main()
{
  int cv;
  cout<<"=====MENU LUA CHON CONG VIEC====="<<endl;
  cout<<"1.Nhap danh sach"<<endl;
  cout<<"2.In danh sach"<<endl;
  cout<<"3.Them phan tu vao danh sach"<<endl;
  cout<<"4.Xoa phan tu khoi danh sach"<<endl;
  cout<<"5.Tim phan tu danh sach"<<endl;
  cout<<"6.Thoat chuong trinh"<<endl;
  cout<<"================================"<<endl;
  cout<<"Chon cong viec can lam:";
  cin>>cv;
  switch(cv)
  {
    case 1: cout<<"Nhap danh sach"<<endl;break;
    case 2: cout<<"In danh sach"<<endl;break;
    case 3: cout<<"Them pt vao danh sach"<<endl;break;
    case 4: cout<<"Xoa pt khoi danh sach"<<endl;break;
    case 5: cout<<"Tim phan tu danh sach"<<endl;break;
    case 6: cout<<"Thoat chuong trinh"<<endl;break;
    default: cout<<"Ban chon cv khong co trong menu";
  }
}
```

1.2. Cấu trúc lặp

1.2.1. Cấu trúc for

Cú pháp

```
for(dãy biểu thức 1; điều kiện lặp; dãy biểu thức 2)
     khối lệnh lặp;
```

- Các biểu thức trong các dãy biểu thức 1, 2 cách nhau bởi dấu phẩy (,). Có thể có nhiều biểu thức trong các dãy này hoặc dãy biểu thức cũng có thể trống.

- Điều kiện lặp: là biểu thức logic (có giá trị đúng hoặc sai).

✧ Các dãy biểu thức và điều kiện có thể trống tuy nhiên vẫn giữ lại các dấu chấm phẩy (;) để ngăn cách các thành phần với nhau.

Cách thực hiện

Khi gặp câu lệnh for trình tự thực hiện của chương trình như sau:

- Thực hiện dãy biểu thức 1 (thông thường là các lệnh khởi tạo cho các biến).

- Kiểm tra điều kiện lặp, nếu đúng thì thực hiện khối lệnh lặp → thực hiện dãy biểu thức 2 → quay lại kiểm tra điều kiện lặp và lặp lại quá trình trên cho đến khi việc kiểm tra điều kiện lặp cho kết quả sai thì dừng.

Tóm lại, dãy biểu thức 1 sẽ được thực hiện 1 lần duy nhất ngay từ đầu quá trình lặp sau đó thực hiện các câu lệnh trong khối lệnh lặp và dãy biểu thức 2 cho đến khi nào không còn thỏa điều kiện lặp nữa thì dừng.

Tương tự như cấu trúc if, nếu khối lệnh lặp có nhiều hơn 1 câu lệnh thì phải bao chúng trong cặp ngoặc móc { }.

Đặc điểm

Thông qua phần giải thích cách hoạt động của câu lệnh for ta có thể nhận xét rằng các thành phần của for có thể để trống, tuy nhiên các dấu chấm phẩy vẫn phải giữ lại để ngăn cách các thành phần với nhau.

Lệnh for lồng nhau

Trong khối lệnh lặp có thể chứa cả lệnh for, tức các lệnh for cũng được phép lồng nhau như các lệnh có cấu trúc khác.

1.2.2. Cấu trúc while

Cú pháp

```
while(điều kiện)
     khối lệnh lặp;
```

Cách thực hiện

Khi gặp lệnh while chương trình thực hiện như sau: đầu tiên chương trình sẽ kiểm tra điều kiện, nếu đúng thì thực hiện khối lệnh lặp, sau đó quay lại kiểm tra điều kiện

và tiếp tục, nếu điều kiện sai thì dừng vòng lặp. Tóm lại có thể mô tả một cách ngắn gọn về câu lệnh while như sau: lặp lại khối lệnh trong khi điều kiện vẫn còn đúng.

Đặc điểm

- Khối lệnh lặp có thể không được thực hiện lần nào nếu điều kiện sai ngay từ đầu.

- Để vòng lặp không lặp vô tận thì trong khối lệnh thông thường phải có ít nhất một câu lệnh nào đó gây ảnh hưởng đến kết quả của điều kiện, tức là làm cho điều kiện đang đúng trở thành sai.

- Nếu điều kiện luôn luôn nhận giá trị đúng (ví dụ biểu thức điều kiện là 1) thì trong khối lệnh lặp phải có câu lệnh kiểm tra dừng và lệnh break.

- Nếu khối lệnh lặp có nhiều hơn 1 câu lệnh thì phải bao chúng trong cặp ngoặc móc.

1.2.3. Cấu trúc do ... while

Cú pháp

```
do
    khối lệnh lặp;
while(điều kiện);
```

Cách thực hiện

Đầu tiên chương trình sẽ thực hiện khối lệnh lặp, tiếp theo kiểm tra điều kiện, nếu điều kiện còn đúng thì quay lại thực hiện khối lệnh và quá trình tiếp tục cho đến khi điều kiện trở thành sai thì dừng.

Đặc điểm

Các đặc điểm của câu lệnh do … while cũng giống với câu lệnh lặp while trừ điểm khác biệt, đó là khối lệnh trong do … while sẽ được thực hiện ít nhất một lần vì điều kiện được kiểm tra sau, trong khi khối lệnh trong while có thể không được thực hiện lần nào.

Chương trình minh họa 3

Viết chương trình nhập vào mảng 1 chiều (danh sách đặc) gồm n phần tử kiểu số nguyên (n không quá 100). In ra danh sách đã nhập. Tìm và trả về vị trí xuất hiện đầu tiên của x. Thêm 1 phần tử vào đầu danh sách. Xóa phần tử ở chính giữa danh sách. Sắp xếp danh sách theo thứ tự tăng dần.

```
#include <iostream>
using namespace std;
main()
{
     int ds[100]
     int n, i, j, x, tam;
```

```
//Nhap va kiem tra so phan tu cua danh sach
do
{  cout<<"Nhap so phan tu n = ";
   cin>>n;
} while(n <= 0 || n > 100);
//Nhap danh sach
for(i=0; i<n; i++)
{
   cout<<"Nhap phan tu thu "<<i+1<<" : ";
   cin>>ds[i];
}
//In danh sach
cout<<"Danh sach vua nhap la :"<<endl;
for(i=0; i<n; i++)
   cout<<ds[i]<<'\t';
cout<<endl;
//Tim x
cout<<"Nhap phan tu cam tim: "; cin>>x;
i=0;
while(i<n && ds[i]!=x)
    i++;
if(i<n) cout<<x<<" x.hien d.tien o v.tri"<<i<<endl;
else cout<<"Khong co "<<x<<" trong danh sach"<<endl;
//Them phan tu vao dau danh sach
cout<<"Nhap noi dung phan tu can them: ";
cin>>x;
for(i=n-1;i>=0;i--)
   ds[i+1]=ds[i];
ds[0]=x;
n=n+1;
cout<<"Danh sach sau khi them la :"<<endl;
for(i=0; i<n; i++)
   cout<<ds[i]<<'\t';
cout<<endl;
```

```
    //Xoa phan tu o chinh giua danh sach
    int vtg=n/2;
    for(i=vtg; i<n-1; i++)
       ds[i]=ds[i+1];
    n=n-1;
    ds[n]=0;
    cout<<"Danh sach sau khi xoa la :"<<endl;
    for(i=0; i<n; i++)
       cout<<ds[i]<<'\t';
    cout<<endl;
    //Sap xep danh sach bang phuong phap Bubble Sort
    for(i=0; i<n-1; i++)
        for(j=n-1; j>i; j--)
            if(ds[j]<ds[j-1])
            {   tam=ds[j];
                ds[j]=ds[j-1];
                ds[j-1]=tam;
            }
    cout<<"Danh sach sau khi sap xep la :"<<endl;
    for(i=0; i<n; i++)
       cout<<ds[i]<<'\t';
    cout<<endl;
}
```

2. CON TRỎ – HÀM – ĐỆ QUI

2.1. Con trỏ

2.1.1. Định nghĩa

Con trỏ là một biến dùng để giữ địa chỉ của biến khác. Nếu p là con trỏ giữ địa chỉ của biến x ta gọi p trỏ tới x và x được trỏ bởi p. Thông qua con trỏ ta có thể làm việc được với nội dung của những ô nhớ mà p trỏ đến.

Con trỏ là một đặc trưng mạnh của C++, nó cho phép chúng ta thâm nhập trực tiếp vào bộ nhớ để xử lý các bài toán khó chỉ bằng vài câu lệnh đơn giản của chương trình.

2.1.2. Khai báo

<Kiểu được trỏ> <*Tên biến con trỏ> ;

<Kiểu được trỏ*> <Tên biến con trỏ> ;

Địa chỉ của một biến là địa chỉ byte nhớ đầu tiên của biến đó. Vì vậy để lấy được nội dung của biến, con trỏ phải biết được số byte của biến, tức là kiểu của biến mà con trỏ sẽ trỏ tới. Kiểu này cũng được gọi là kiểu của con trỏ. Như vậy khai báo biến con trỏ cũng giống như khai báo một biến thường ngoại trừ cần thêm dấu * trước tên biến (hoặc sau tên kiểu).

Ví dụ:

int *p ; // khai báo biến p là biến con trỏ trỏ đến kiểu số nguyên.

float *q, *r ; // khai báo 2 con trỏ thực q và r.

2.1.3. Cách sử dụng

- Để con trỏ p trỏ đến biến x ta phải dùng phép gán p = địa chỉ của x. Nếu x không phải là biến mảng thì ta viết: p = &x. Còn nếu x là biến mảng thì ta viết: p = x hoặc p = &x[0].

- Phép toán * cho phép lấy nội dung nơi p trỏ đến, ví dụ để gán nội dung nơi p trỏ đến cho biến f ta viết f = *p.

- Ta không thể gán p cho một hằng địa chỉ cụ thể. Chẳng hạn ta viết p = 200 là sai.

- Phép toán & và phép toán * là 2 phép toán ngược nhau. Cụ thể nếu p = &x thì x = *p. Từ đó nếu p trỏ đến x thì bất kỳ nơi nào xuất hiện x đều có thể thay được bởi *p và ngược lại.

2.1.4. Các phép toán

Phép toán gán

- Gán con trỏ với địa chỉ một biến: p = &x ;

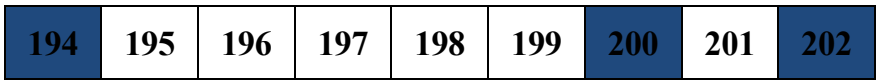
- Gán con trỏ với con trỏ khác: p = q ; (sau phép toán gán này p, q chứa cùng một địa chỉ hay cùng trỏ đến một nơi).

Phép toán tăng, giảm địa chỉ

p ± n: con trỏ trỏ đến thành phần thứ n sau (trước) p.

Một đơn vị tăng giảm của con trỏ bằng kích thước của biến được trỏ.

Ví dụ: Giả sử p là con trỏ nguyên (2 bytes) đang trỏ đến địa chỉ 200 thì p+1 là con trỏ trỏ đến địa chỉ 202; tương tự, p + 5 là con trỏ trỏ đến địa chỉ 210; và p − 3 chứa địa chỉ 194.



p - 3 p p + 1

Như vậy, phép toán tăng, giảm con trỏ cho phép làm việc thuận lợi trên mảng. Nếu con trỏ đang trỏ đến mảng (tức đang chứa địa chỉ đầu tiên của mảng), việc tăng con trỏ lên 1 đơn vị sẽ dịch chuyển con trỏ trỏ đến phần tử thứ hai, và cứ thế tiếp tục. Từ đó ta có thể cho con trỏ chạy từ đầu đến cuối mảng bằng cách tăng con trỏ lên từng đơn vị như trong câu lệnh for trong ví dụ dưới đây.

Phép toán tự tăng, tự giảm

p++, p--, ++p, --p: Tương tự p + 1 và p - 1, có chú ý đến tăng (giảm) trước, sau.

Ví dụ sau minh họa kết quả kết hợp phép tự tăng, tự giảm với việc lấy giá trị nơi con trỏ trỏ đến. a là một mảng gồm 2 số, p là con trỏ trỏ đến mảng a. Các lệnh dưới đây được qui ước là độc lập với nhau (tức lệnh sau không bị ảnh hưởng bởi lệnh trước, đối với mỗi lệnh p luôn luôn trỏ đến phần tử đầu (a[0]) của a.

Ví dụ:

```
#include <iostream>
using namespace std;
main()
{    int a[2] = {3, 7}, *p = a;
     cout << (*p)++ << endl; // in ra 3
     cout << *p << endl; // in ra 4
     cout << ++(*p) << endl; // in ra 5
     cout << *p << endl; // in ra 5
     cout << *(p++) << endl ; // in ra 5
     cout << *p << endl; // in ra 7
}
```

✧ ***Chú ý***: Ta phải phân biệt giữa p + 1 và p++ (hoặc ++p):

• p + 1 được xem như một con trỏ khác với con trỏ p. Con trỏ p + 1 trỏ đến phần tử sau p.

• p++ là con trỏ p nhưng trỏ đến phần tử khác. Con trỏ p++ trỏ đến phần tử đứng sau phần tử p.

Phép hiệu của 2 con trỏ

Phép toán này chỉ thực hiện được khi p và q là 2 con trỏ cùng trỏ đến các phần tử của một dãy dữ liệu nào đó trong bộ nhớ (chẳng hạn cùng trỏ đến 1 mảng dữ liệu).

Khi đó hiệu p - q là số phần tử giữa p và q (chú ý p - q không phải là hiệu của 2 địa chỉ mà là số phần tử giữa p và q).

Ví dụ:

Giả sử p và q là 2 con trỏ số nguyên có kích thước 2 bytes, p có địa chỉ 200 và q có địa chỉ 208. Khi đó p - q = - 4 và q - p = 4 (4 là số phần tử nguyên từ địa chỉ 200 đến 208).

Phép toán so sánh

Các phép toán so sánh cũng được áp dụng đối số với con trỏ, thực chất là so sánh giữa địa chỉ của hai nơi được trỏ bởi các con trỏ này. Thông thường các phép so sánh <, <=, >, >= chỉ áp dụng cho hai con trỏ trỏ đến phần tử của cùng một mảng dữ liệu nào đó. Thực chất của phép so sánh này chính là so sánh chỉ số của 2 phần tử được trỏ bởi 2 con trỏ đó.

2.1.5. Cấp phát động, toán tử cấp phát và thu hồi vùng nhớ

Cấp phát động

Khi tiến hành chạy chương trình, chương trình dịch sẽ bố trí các ô nhớ cụ thể cho các biến được khai báo trong chương trình. Vị trí cũng như số lượng các ô nhớ này tồn tại và cố định trong suốt thời gian chạy chương trình, chúng xem như đã bị chiếm dụng và sẽ không được sử dụng vào mục đích khác và chỉ được giải phóng sau khi chấm dứt chương trình. Việc phân bổ bộ nhớ như vậy được gọi là ***cấp phát tĩnh*** (vì được cấp sẵn trước khi chạy chương trình và không thể thay đổi tăng, giảm kích thước hoặc vị trí trong suốt quá trình chạy chương trình).

Ví dụ nếu ta khai báo một mảng nguyên chứa 1000 phần tử thì trong bộ nhớ sẽ có một vùng nhớ liên tục 2000 bytes để chứa dữ liệu của mảng này. Khi đó dù trong chương trình ta chỉ nhập vào mảng và làm việc với một vài phần tử thì phần mảng rỗi còn lại vẫn không được sử dụng vào việc khác. Đây là hạn chế thứ nhất của kiểu mảng. Ở một hướng khác, một lần nào đó chạy chương trình ta lại cần làm việc với hơn 1000 số nguyên. Khi đó vùng nhớ mà chương trình dịch đã dành cho mảng là không đủ để sử dụng. Đây chính là hạn chế thứ hai của mảng được khai báo trước.

Để khắc phục các hạn chế trên của kiểu mảng, bây giờ chúng ta sẽ không khai báo (bố trí) trước mảng dữ liệu với kích thước cố định như vậy. Kích thước cụ thể sẽ được cấp phát trong quá trình chạy chương trình theo đúng yêu cầu của NSD. Nhờ vậy chúng ta có đủ số ô nhớ để làm việc mà vẫn tiết kiệm được bộ nhớ, và khi không dùng nữa ta có thể thu hồi (còn gọi là giải phóng) số ô nhớ này để chương trình sử dụng vào việc khác. Hai công việc cấp phát và thu hồi này được thực hiện thông qua các toán tử new và delete, để thực hiện được điều này ta cần phải sử dụng biến kiểu con trỏ. Thông qua biến con trỏ ta có thể làm việc với bất kỳ địa chỉ nào của vùng nhớ được cấp phát. Cách thức bố trí bộ nhớ như thế này được gọi là ***cấp phát động***.

Toán tử cấp phát vùng nhớ (new)

Cú pháp:

p = new <kiểu>;

p = new <kiểu>[n];

Với p là biến con trỏ.

Ví dụ:

```
int *p;
p = new int;
p = new int[10];
```

Khi gặp toán tử new, chương trình sẽ tìm trong bộ nhớ một lượng ô nhớ còn rỗi và liên tục với số lượng đủ theo yêu cầu và cho p trỏ đến địa chỉ (byte đầu tiên) của vùng nhớ này. Nếu không có vùng nhớ với số lượng như vậy thì việc cấp phát là

thất bại và p = NULL (NULL là một địa chỉ rỗng, không xác định). Do vậy ta có thể kiểm tra việc cấp phát có thành công hay không thông qua việc kiểm tra con trỏ p bằng hay khác NULL.

Toán tử thu hồi vùng nhớ (delete)

Để thu hồi hay giải phóng vùng nhớ đã cấp phát cho một biến (khi không cần sử dụng nữa) ta sử dụng toán tử delete.

Cú pháp:

delete p;

Với p là con trỏ được cấp phát vùng nhớ bởi toán tử new.

Để giải phóng toàn bộ mảng được cấp pháp thông qua con trỏ p ta dùng câu lệnh:

delete[] p;

Với p là con trỏ trỏ đến mảng.

Chương trình minh họa 4

Viết chương trình nhập vào dãy số nguyên (không dùng mảng). In ra dãy đã nhập. Sắp xếp dãy tăng dần và in ra dãy kết quả.

Trong ví dụ này chương trình xin cấp phát bộ nhớ đủ chứa n số nguyên và được trỏ bởi con trỏ head. Khi đó địa chỉ của số nguyên đầu tiên và cuối cùng sẽ là head và head + n - 1. p và q là 2 con trỏ chạy trên dãy số này, so sánh và đổi nội dung của các số này với nhau để sắp thành dãy tăng dần và cuối cùng in kết quả.

```
#include <iostream>
using namespace std;
main()
{    int *head, *p, *q, n, tam;
                  // head trỏ đến (đánh dấu) đầu dãy
     cout << "Cho biet so phan tu cua day: "; cin >> n;
     head = new int[n];//cấp phát bộ nhớ chứa n số nguyên
     for(p = head; p < head + n; p++) // nhập dãy
         {    cout<<"Nhap p.tu thu " << p-head+1 << ": ";
              cin >> *p ;
         }
     cout << "Day vua nhap la : " << endl;
     for(p = head; p < head + n; p++)
          cout << *p << '\t';
    cout << endl;
```

```
     // Sắp xếp dãy tăng dần
    for(p = head; p < head + n - 1; p++)
          for (q = p + 1; q < head + n; q++)
              if (*q < *p)
                 {    tam = *p;
                      *p = *q;
                      *q = tam;
                 }
     cout << "Day sau khi sap xep la : " << endl;
    for(p = head; p < head + n; p++)
        cout << *p << '\t';
    cout << endl;
}
```

2.2. Hàm

2.2.1. Định nghĩa

Hàm (Function) là một chương trình con trong chương trình lớn. Hàm nhận (hoặc không) các đối số (tham số) và trả lại (hoặc không) một giá trị cho chương trình gọi nó. Trong trường hợp không trả lại giá trị, hàm hoạt động như một thủ tục trong các NNLT khác. Một chương trình là một tập hợp các hàm, trong đó có một hàm chính với tên gọi là main(), khi chạy chương trình, hàm main() sẽ được chạy đầu tiên và gọi đến các hàm khác. Kết thúc hàm main() cũng là kết thúc chương trình.

Hàm giúp cho việc phân đoạn chương trình thành những mô-đun riêng lẻ, hoạt động độc lập với ngữ nghĩa của chương trình lớn, có nghĩa là một hàm có thể được sử dụng trong chương trình này mà cũng có thể được sử dụng trong chương trình khác, dễ dàng cho việc kiểm tra và bảo trì chương trình.

2.2.2. Các đặc trưng

- Nằm trong hoặc ngoài văn bản có chương trình gọi đến hàm, trong một văn bản có thể chứa nhiều hàm.

- Được gọi từ chương trình chính (hàm main()), từ hàm khác hoặc từ chính nó (tính đệ qui).

- Không lồng nhau.

2.2.3. Khai báo

Một hàm thường thực hiện chức năng: tính toán trên các đối số và trả lại kết quả, hoặc chỉ đơn thuần thực hiện một chức năng nào đó, không trả lại kết quả tính toán. Thông thường kiểu của kết quả trả về được gọi là kiểu của hàm.

Các hàm thường được khai báo ở đầu chương trình. Các hàm viết sẵn được khai báo trong các tệp nguyên mẫu (tập tin tiêu đề) *.h. Do đó, để sử dụng được các hàm này, cần có chỉ thị `#include <*.h>` ở ngay đầu chương trình, trong đó *.h là tên file cụ thể có chứa khai báo của các hàm được sử dụng (chẳng hạn để sử dụng các hàm toán học ta cần khai báo `#include <math.h>`). Đối với các hàm do người lập trình tự viết, cũng cần phải khai báo.

Khai báo một hàm như sau:

<Kiểu giá trị trả về> <Tên hàm> (Danh sách đối số) ;

Trong đó, kiểu giá trị trả về còn gọi là kiểu của hàm và có thể nhận bất kỳ kiểu chuẩn nào của C++ và cả kiểu do người lập trình khai báo. Đặc biệt nếu hàm không trả về giá trị thì kiểu giá trị trả về được khai báo là **void**.

Ví dụ: long LapPhuong(int);

int NgauNhien();

void ChuoiHoa(char[ ]);

int Cong(int, int);

2.2.4. Cấu trúc chung

Cấu trúc chung của một hàm bất kỳ được bố trí cũng giống như hàm main() trong các phần trước.

<Kiểu giá trị trả về> <Tên hàm> (Danh sách đối số hình thức)

{

Các khai báo cục bộ của hàm ; // chỉ dùng riêng cho hàm này

Dãy lệnh của hàm ; // các câu lệnh xử lý

return <Biểu thức trả về> ; // có thể nằm đâu đó trong dãy lệnh.

}

- **Danh sách đối số hình thức** còn được gọi ngắn gọn là danh sách đối số gồm dãy các đối số cách nhau bởi dấu phẩy, đối số có thể là một biến thường, biến tham chiếu hoặc biến con trỏ. Mỗi đối số được khai báo giống như khai báo biến, tức là một cặp gồm **<kiểu đối số> <tên đối số>**.

- Với **hàm có trả lại giá trị** cần có câu lệnh return kèm theo sau là một biểu thức. Kiểu của giá trị biểu thức này chính là kiểu của hàm đã được khai báo ở phần tên hàm. Câu lệnh return có thể nằm ở vị trí bất kỳ trong phần dãy lệnh của hàm. Khi gặp câu lệnh return chương trình tức khắc thoát khỏi hàm và trả lại giá trị của biểu thức sau return như giá trị của hàm.

- Với **hàm không có trả lại giá trị** (tức kiểu hàm là void) thì không cần có câu lệnh return hoặc có câu lệnh return nhưng phía sau return không có <Biểu thức trả về>.

Chương trình minh họa 5

Xây dựng hàm tính lập phương của số nguyên n (với n nguyên) và hàm đổi chuỗi bất kỳ về dạng chuẩn upper. Viết hàm main() để kiểm tra tính đúng đắn của các hàm.

```
#include <iostream>
#include <string.h>
using namespace std;
long LapPhuong(int);
void ChuoiHoa(char *s);
main()
{   int n;
    cout<<"Nhap so nguyen n = "; cin>>n;
    cout<<"Lap phuong cua "<<n<<" la : ";
    cout<<LapPhuong(n)<<endl;
    cin.ignore(1);
    char *s = new char[100];
    cout<<"Nhap vao chuoi bat ky : ";
    cin.getline(s, 100);
    ChuoiHoa(s);
    cout<<"Chuoi dang upper la : "<<s<<endl;
}
long LapPhuong(int n)
{
    return n*n*n;
}
void ChuoiHoa(char *s)
{
    for(int i = 0; i < strlen(s); i++)
        s[i] = toupper(s[i]);
}
```

✧ ***Các chú ý***:

- Danh sách đối số trong khai báo hàm có thể chứa hoặc không chứa tên đối số, thông thường ta chỉ khai báo kiểu đối số chứ không cần khai báo tên đối số, trong khi ở hàng đầu tiên của định nghĩa hàm phải có tên đối số đầy đủ.

- Cuối khai báo hàm phải có dấu chấm phẩy (;), trong khi cuối hàng đầu tiên của định nghĩa hàm không có dấu chấm phẩy.

- Hàm có thể không có đối số(danh sách đối số rỗng), tuy nhiên cặp dấu ngoặc sau tên hàm vẫn phải được viết. Chẳng hạn như: NgauNhien(), InTho(), …

- Một hàm có thể không cần phải khai báo nếu nó được định nghĩa trước khi có hàm nào đó gọi đến nó.

2.2.5. Lời gọi hàm

Lời gọi hàm được phép xuất hiện trong bất kỳ biểu thức, câu lệnh của hàm khác … Nếu lời gọi hàm lại nằm trong chính bản thân hàm đó thì ta gọi là đệ qui. Để gọi hàm ta chỉ cần viết tên hàm và danh sách các giá trị cụ thể truyền cho các đối số đặt trong cặp dấu ngoặc ().

Tên hàm(danh sách đối số thực tế);

- Danh sách đối số (tham số) thực tế còn gọi là danh sách giá trị gồm các giá trị cụ thể để gán lần lượt cho các đối số hình thức của hàm. Khi hàm được gọi thực hiện thì tất cả những vị trí xuất hiện của đối số hình thức sẽ được gán cho giá trị cụ thể của đối số thực tế tương ứng trong danh sách, sau đó hàm tiến hành thực hiện các câu lệnh của hàm (để tính kết quả).

- Danh sách đối số thực tế truyền cho đối số hình thức có số lượng bằng với số lượng đối số trong hàm và được truyền cho đối số theo thứ tự tương ứng. Các đối số thực tế có thể là các hằng, các biến hoặc biểu thức. Biến trong giá trị có thể trùng với tên đối số. Ví dụ ta có hàm in n lần ký tự c với khai báo `void inkitu(int n, char c);` và lời gọi hàm `inkitu(12, 'A');` thì n và c là các đối số hình thức, 12 và 'A' là các đối số thực tế hoặc giá trị. Các đối số hình thức n và c sẽ lần lượt được gán bằng các giá trị tương ứng là 12 và 'A' trước khi tiến hành các câu lệnh trong phần thân hàm. Giả sử hàm in ký tự được khai báo lại thành `inkitu(char c, int n);` thì lời gọi hàm cũng phải được thay lại thành `inkitu('A', 12);`

- Các giá trị tương ứng được truyền cho đối số phải có kiểu cùng với kiểu đối số (hoặc C++ có thể tự động chuyển về kiểu của đối số).

- Khi một hàm được gọi, nơi gọi tạm thời chuyển điều khiển đến thực hiện câu lệnh đầu tiên trong hàm được gọi. Sau khi kết thúc thực hiện hàm, điều khiển lại được trả về thực hiện tiếp câu lệnh sau lệnh gọi hàm của nơi gọi.

Chương trình minh họa 6

Viết chương trình tính giá trị hàm $f = 2x^3 - 5x^2 - 4x + 1$, thay cho việc tính trực tiếp x^3 và x^2, ta xây dựng và gọi hàm LuyThua() để tính các giá trị này trong hàm main().

```
#include <iostream>
using namespace std;
double LuyThua(float x, int n)
{   int i;
    double kq = 1;
```

```
    for (i = 1; i <= n; i++)
        kq *= x;
    return kq;
}
main()
{
    float x;
    double f;
    cout << "Nhap x = ";
    cin >> x;
    f = 2*LuyThua(x,3) - 5*LuyThua(x,2) - 4*x + 1;
    cout << "Gia tri cua ham f = " << f << endl;
}
```

2.2.6. Hàm với đối số mặc định

Trong phần trước chúng ta đã khẳng định số lượng đối số thực tế phải bằng số lượng đối số hình thức khi gọi hàm. Tuy nhiên, trong thực tế rất nhiều lần hàm được gọi với các giá trị của một số đối số hình thức được lặp đi lặp lại. Trong trường hợp như vậy lúc nào cũng phải viết một danh sách dài các đối số thực tế giống nhau cho mỗi lần gọi là một công việc không mấy thú vị. Từ thực tế đó C++ đưa ra một cú pháp
mới về hàm sao cho một danh sách đối số thực tế trong lời gọi không nhất thiết phải viết đầy đủ nếu một số trong chúng đã có sẵn những giá trị định trước. Cú pháp này được gọi là hàm với đối số mặc định và được khai báo với cú pháp như sau:

<Kiểu hàm> <Tên hàm>(đs1, …, đsn, đsmđ1 = gt1, …, đsmđm = gtm) ;

- Các đối số đs1, …, đsn và các đối số mặc định đsmđ1, …, đsmđm đều được khai báo như cũ nghĩa là gồm có kiểu đối số và tên đối số.

- Riêng các đối số mặc định đsmđ1, …, đsmđm có gán thêm các giá trị mặc định gt1, …, gtm. Một lời gọi bất kỳ khi gọi đến hàm này đều phải có đầy đủ các đối số thực tế ứng với các đs1, …, đsn nhưng có thể có hoặc không các đối số thực tế ứng với các đối số mặc định đsmđ1, …, đsmđm. Nếu đối số nào không có đối số thực tế thì nó sẽ được tự động gán giá trị mặc định đã khai báo.

Ví dụ: Xét hàm `double LuyThua(float x, int n = 2);` Hàm này có một đối số mặc định là số mũ n, nếu lời gọi hàm bỏ qua số mũ này thì chương trình hiểu là tính bình phương của x (n = 2). Chẳng hạn lời gọi `LuyThua(4, 3)` được hiểu là tính 4^3, lời gọi `LuyThua(4, 2)` hay `LuyThua(4)` được hiểu là tính 4^2

✧ ***Lưu ý***: Các đối số mặc định phải nằm bên phải các đối số không mặc định.

2.2.7. Các cách truyền đối số

Có 3 cách truyền đối số thực tế cho các đối số hình thức trong lời gọi hàm. Trong đó cách ta đã dùng cho đến thời điểm hiện nay được gọi là truyền bằng giá trị, còn được gọi là *tham trị*, tức các đối số hình thức sẽ nhận các giá trị cụ thể từ lời gọi hàm và tiến hành tính toán rồi trả lại giá trị.

Truyền bằng giá trị

Ta xét lại ví dụ hàm `LuyThua(float x, int n)` dùng để tính tính x^n. Giả sử trong chương trình chính (hàm main()) ta có các biến a, b, f đang chứa các giá trị a = 2, b = 3, và f chưa có giá trị. Để tính a^b và gán giá trị trả về vào f, ta có thể gọi `f = LuyThua(a,b);`

Khi gặp lời gọi này, chương trình sẽ tổ chức như sau:

- Tạo 2 biến mới (tức 2 ô nhớ trong bộ nhớ) có tên x và n. Gán nội dung các ô nhớ này bằng các giá trị trong lời gọi, tức gán 2 (a) cho x và 3 (b) cho n.

- Tới phần khai báo (của hàm), chương trình tạo thêm các ô nhớ mang tên là i và kq.

- Tiến hành tính toán (gán lại kết quả cho kq).

- Cuối cùng lấy kết quả trong kq gán cho ô nhớ f (là ô nhớ có sẵn đã được khai báo trước, nằm bên ngoài hàm).

- Kết thúc hàm quay về chương trình gọi. Do hàm LuyThua đã hoàn thành xong việc tính toán nên các ô nhớ được tạo ra trong khi thực hiện hàm (x, n, i, kq) sẽ được xóa khỏi bộ nhớ. Kết quả tính toán được lưu giữ trong ô nhớ f (không bị xóa vì không liên quan gì đến hàm).

Truyền đối số theo giá trị là cách truyền phổ biến. Tuy nhiên vấn đề đặt ra là giả sử ngoài việc tính f, ta còn muốn thay đổi các giá trị của các ô nhớ a, b (khi truyền nó cho hàm) thì có thể thực hiện được không? Để giải quyết vấn đề này ta cần phải truyền đối số bằng địa chỉ hoặc con trỏ. Hai cách truyền này có thể được gọi chung là *tham biến*.

Truyền bằng con trỏ

Xét ví dụ hoán đổi giá trị của 2 biến. Đây là một yêu cầu nhỏ nhưng được gặp nhiều lần trong chương trình, ví dụ để sắp xếp một mảng hay danh sách. Do vậy cần viết một hàm để thực hiện yêu cầu trên. Hàm không trả kết quả mà chỉ hoán đổi giá trị giữa 2 đối số. Do các biến cần hoán đổi là chưa được biết trước tại thời điểm viết hàm, nên ta phải đưa chúng vào hàm như các đối số, tức là hàm có 2 đối số (có thể đặt là x, y) đại diện cho các biến sẽ thay đổi giá trị sau này.

Từ các nhận xét trên, nếu hàm hoán đổi ngay sau đây sẽ không đáp ứng được yêu cầu.

```
void Swap(int x, int y)
{
    int t = x;
    x = y; y = t;
}
```

Hàm Swap trên không đáp ứng được yêu cầu vì giả sử trong chương trình chính ta có 2 biến x, y chứa các giá trị lần lượt là 2 và 5. Ta cần hoán đổi nội dung 2 biến này sao cho x = 5 còn y = 2 bằng cách gọi đến hàm Swap(x, y).

```
main()
{   int x = 2, y = 5;
    Swap(x, y);
    cout << "x = " << x << "va y = " << y << endl;
            // in ra x = 2 va y = 5 (x, y vẫn không đổi)
}
```

Thực tế sau khi chạy xong chương trình ta thấy giá trị của x và y vẫn không thay đổi!? Như đã giải thích trong mục trên (gọi hàm LuyThua), việc đầu tiên khi chương trình thực hiện một hàm là tạo ra các biến mới (các ô nhớ mới, độc lập với các ô nhớ x, y đã có sẵn) tương ứng với các đối số, trong trường hợp này cũng có tên là x, y và gán nội dung của x, y (ngoài hàm) cho x, y (mới). Và việc cuối cùng của chương trình sau khi thực hiện xong hàm là xóa các biến mới này. Do vậy nội dung của các biến mới thực tế là có thay đổi, nhưng không ảnh hưởng gì đến các biến x, y cũ (ngoài hàm).

Như vậy ***hàm Swap cần được viết lại*** sao cho việc thay đổi giá trị không thực hiện trên các biến tạm mà phải thực hiện trên các biến ngoài. Muốn vậy thay vì truyền giá trị của các biến ngoài cho đối số, bây giờ ta sẽ truyền địa chỉ của nó cho đối số, và các thay đổi sẽ phải thực hiện trên nội dung của các địa chỉ này. Đó chính là lý do ta phải sử dụng con trỏ để làm đối số thay cho biến thường.

Cụ thể hàm Swap được viết lại như sau:

```
void Swap(int *p, int *q)
{
    int t;
    t = *p; *p = *q; *q = t;
}
```

Với cách tổ chức hàm như vậy rõ ràng nếu ta cho p trỏ tới biến x và q trỏ tới biến y thì hàm Swap sẽ làm thay đổi nội dung của x, y chứ không phải của p, q.

Từ đó lời gọi hàm sẽ là Swap(&x, &y) (tức truyền địa chỉ của x cho p để p trỏ tới x và tương tự q trỏ tới y).

Truyền bằng địa chỉ

Một hàm viết dưới dạng đối số được truyền bằng địa chỉ sẽ đơn giản hơn so với con trỏ và nó giống với cách truyền bằng giá trị hơn, trong đó chỉ có một điểm khác biệt đó là các đối số khai báo theo dạng địa chỉ.

Như vậy ***hàm Swap trên cũng có thể được viết*** theo cách truyền địa chỉ trực tiếp mà không thông qua con trỏ như sau:

```
void Swap(int &x, int &y)
{
    int t = x;
    x = y; y = t;
}
```

Và lời gọi hàm cũng đơn giản như cách truyền đối số bằng giá trị.

```
main()
{   int x = 2, y = 5;
    Swap(x, y);
    cout << "x = " << x << "va y = " << y << endl;
            // in ra x = 5 va y = 2 (x, y đã không đổi)
}
```

2.3. Đệ qui

2.3.1. Khái niệm đệ qui

Một hàm gọi đến hàm khác là việc gọi hàm một cách thông thường, tuy nhiên nếu một hàm lại gọi đến chính bản thân nó thì ta gọi hàm là đệ qui. Khi thực hiện một hàm đệ qui, hàm sẽ phải chạy rất nhiều lần, trong mỗi lần chạy chương trình sẽ tạo nên một tập biến cục bộ mới trên ngăn xếp (các đối số, các biến riêng khai báo trong hàm) độc lập với lần chạy trước đó, từ đó dễ gây tràn ngăn xếp. Vì vậy đối với những bài toán có thể giải được bằng phương pháp lặp thì không nên dùng đệ qui.

Để minh họa ta hãy xét hàm tính n giai thừa. Để tính n! ta có thể dùng phương pháp lặp như sau:

```
long GiaiThua(int n)
{    long kq = 1;
     for (int i = 1; i <= n; i++)
        kq = kq * i;
     return kq;
}
```

Mặt khác, n! cũng được tính thông qua (n-1)! bởi công thức truy hồi như sau:

n! = 1 nếu n = 0

n! = (n-1)! * n nếu n > 0

Do đó ta có thể xây dựng hàm đệ qui tính n! như sau:

```
long GiaiThuaDQ(int n)
{    if(n == 0)
         return 1;
     else
         return GiaiThuaDQ(n - 1) * n;
}
```

2.3.2. Các đặc điểm của hàm đệ qui

- Chương trình viết ngắn gọn.

- Việc thực hiện gọi đi gọi lại hàm rất nhiều lần phụ thuộc vào độ lớn của đầu vào. Chẳng hạn trong ví dụ trên hàm được gọi n lần, mỗi lần như vậy chương trình sẽ mất thời gian để lưu giữ các thông tin của hàm gọi trước khi chuyển điều khiển đến thực hiện hàm được gọi. Mặt khác các thông tin này được lưu giữ nhiều lần trong ngăn xếp sẽ tốn nhiều vùng nhớ và có thể dẫn đến tràn ngăn xếp nếu n lớn.

- Chương trình dễ viết và dễ đọc nhưng có thể khó hiểu. Trên thực tế có nhiều bài toán hầu như tìm một thuật toán lặp cho nó là rất khó trong khi viết theo thuật toán đệ qui thì lại rất dễ dàng.

2.3.3. Lớp các bài toán giải được bằng đệ qui

Phương pháp đệ qui thường được dùng để giải các bài toán có đặc điểm:

- Giải quyết được dễ dàng trong các trường hợp riêng gọi là trường hợp suy biến hay cơ sở, trong trường hợp này hàm được tính bình thường mà không cần gọi lại chính nó.

- Đối với trường hợp tổng quát, bài toán có thể giải được bằng bài toán cùng dạng nhưng với đối số khác có kích thước hay giá trị nhỏ hơn đối số ban đầu. Và sau một số bước hữu hạn biến đổi cùng dạng, bài toán được đưa về trường hợp suy biến.

Như vậy trong trường hợp tính n! nếu n = 0 hàm cho ngay giá trị là 1 mà không cần phải gọi lại chính nó, đây chính là trường hợp suy biến. Trường hợp n > 0 hàm sẽ gọi lại chính nó nhưng với n giảm 1 đơn vị. Việc gọi này được lặp lại cho đến khi n = 0.

Một lớp rất rộng của bài toán dạng này là các bài toán có thể định nghĩa được dưới dạng đệ qui như các bài toán lặp với số bước hữu hạn biết trước, các bài toán UCLN, tháp Hà Nội, … Đặc biệt là các ứng dụng trong cấu trúc dữ liệu và giải thuật.

2.3.4. Cấu trúc chung của hàm đệ qui

Dạng thức chung của một hàm đệ qui như sau:

```
if (trường hợp suy biến)
        { trình bày cách giải  // thường là trả về kết quả }
else    // trường hợp tổng quát
        { gọi lại hàm với đối số "nhỏ/lớn" hơn }
```

2.3.5. Các ví dụ

Ví dụ 1: Tính số hạng thứ n của dãy Fibonacci là dãy f(n) được định nghĩa:

- f(0) = f(1) = 1
- f(n) = f(n - 1) + f(n - 2) với ∀ n ≥ 2

Hàm đệ qui được định nghĩa như sau:

```
long Fibo(int n)
{    long kq;
     if (n==0 || n==1)
        kq = 1;
     else
        kq = Fibo(n-1) + Fibo(n-2);
     return kq;
}
```

Ví dụ 2: Hàm đệ qui để tính $C_n^k = C_{n-1}^{k-1} + C_{n-1}^k; C_n^k = 1 \quad nếu \quad k = 0 \quad hoặc \quad k = n$ được định nghĩa như sau:

```
long Tohop(int n, int k)
{
     if (k==0 || k==n)
        return 1;
     else
        return Tohop(n-1,k-1) + Tohop(n-1,k);
}
```

Chương trình minh họa 7

Xây dựng các hàm nhập, xuất, tính tổng các phần tử trong mảng một chiều bằng giải thuật đệ qui. Sau đó viết hàm main() chứng minh tính đúng đắn của các hàm đã xây dựng.

```
#include <iostream>
using namespace std;
typedef int mang[20];
void Nhap(mang a, int n)
{
    if(n>0)
    {
        cout<<"Nhap phan tu thu "<<n<<" : ";
```

```
        cin>>a[n-1];
        Nhap(a,n-1);
    }
}
void Xuat(mang a, int n, int i)
{
    if(i<n)
    {
        cout<<a[i]<<'\t';
        Xuat(a,n,i+1);
    }
}
long Tong(mang a, int n)
{
    if(n==0) return 0;
    else return a[n-1]+Tong(a,n-1);
}
main()
{    mang a;
     int n;
     cout<<"Nhap so phan tu cua mang: ";
     cin>>n;
     Nhap(a,n);
     cout<<"Mang vua nhap la: ";
     Xuat(a,n,0);
     cout<<endl;
     cout<<"Tong cac phan tu trong mang la: ";
     cout<<Tong(a,n)<<endl;
}
```

BÀI TẬP CHƯƠNG 1

-----ଓ✧ଌ-----

1. Xây dựng các hàm sau:

a. Kiểm tra một số nguyên dương có phải là số chính phương không?

b. Tính tổng N số chính phương đầu tiên.

c. Tính C^k_n =

$$\begin{cases} 1, \text{ nếu } k = 0 \text{ hoặc } k = n \\ C^{k-1}_{n-1} + C^k_{n-1}, \text{ nếu } 0 < k < n \end{cases}$$

d. In ra các số là bội số của K trong N số nguyên dương đầu tiên.

e. In dãy Fibonacci

$$\begin{cases} 1, \text{ nếu } n = 1 \\ 1, \text{ nếu } n = 2 \\ F(n-1) + F(n-2), \text{ nếu } n > 2 \end{cases}$$

2. Xây dựng các hàm sau trên mảng 1 chiều gồm N phần tử kiểu số nguyên:

a. Nhập mảng (không đệ qui và đệ qui).

b. Xuất mảng (không đệ qui và đệ qui).

c. Thêm 1 phần tử vào mảng tại vị trí vt.

d. Xóa các phần tử có nội dung là X.

e. Kiểm tra xem mảng có thứ tự tăng dần không?

3. Xây dựng các hàm sau trên ma trận gồm m hàng, n cột, các phần tử kiểu số thực:

a. Nhập ma trận.

b. In ma trận.

c. Tính tổng 2 ma trận.

d. Tính tích 2 ma trận.

e. Kiểm tra 2 ma trận bằng nhau.

4. Xây dựng các hàm sau bằng 2 cách: không đệ qui và đệ qui

a. Kiểm tra một số nguyên dương có phải là số nguyên tố không?

b. Tìm UCLN của 2 số nguyên dương.

c. Tìm BCNN của 2 số nguyên dương.

d. Sắp xếp mảng theo thứ tự tăng dần.

e. Đếm số từ có trong chuỗi.

f. Đảo từng từ trong chuỗi.

CHƯƠNG 2: TỔNG QUAN VỀ CẤU TRÚC DỮ LIỆU VÀ GIẢI THUẬT

-----ଓ✧ଌ-----

1. VAI TRÒ CỦA CẤU TRÚC DỮ LIỆU VÀ GIẢI THUẬT TRONG CNTT

1.1. Mối liên hệ giữa cấu trúc dữ liệu và giải thuật

1.1.1. Xây dựng cấu trúc dữ liệu

Có thể nói rằng không có một chương trình máy tính nào mà không có dữ liệu để xử lý. Dữ liệu có thể là dữ liệu đưa vào (input data), dữ liệu trung gian hoặc dữ liệu đưa ra (output data). Do vậy, việc tổ chức để lưu trữ dữ liệu phục vụ cho chương trình có ý nghĩa rất quan trọng trong toàn bộ hệ thống chương trình. Việc xây dựng cấu trúc dữ liệu quyết định rất lớn đến chất lượng cũng như công sức của người lập trình trong việc thiết kế và cài đặt chương trình.

1.1.2. Xây dựng giải thuật

Khái niệm giải thuật hay thuật giải mà nhiều khi còn được gọi là thuật toán dùng để chỉ phương pháp hay cách thức (method) để giải quyết vấn đề. Giải thuật có thể được minh họa bằng ngôn ngữ tự nhiên (natural language), bằng lưu đồ (flow chart) hoặc bằng mã giả (pseudo code). Trong thực tế, giải thuật thường được minh họa hay thể hiện bằng mã giả tựa trên một hay một số ngôn ngữ lập trình nào đó (thường là ngôn ngữ mà người lập trình chọn để cài đặt thuật toán), chẳng hạn như C, Pascal,…

Khi đã xác định được cấu trúc dữ liệu thích hợp, người lập trình sẽ bắt đầu tiến hành xây dựng giải thuật tương ứng theo yêu cầu của bài toán đặt ra trên cơ sở của cấu trúc dữ liệu đã được chọn. Để giải quyết một vấn đề có thể có nhiều phương pháp khác nhau, do vậy sự lựa chọn phương pháp phù hợp là một việc mà người lập trình phải cân nhắc và tính toán. Sự lựa chọn này cũng có thể góp phần đáng kể trong việc giảm bớt công việc của người lập trình trong phần cài đặt thuật toán trên một ngôn ngữ cụ thể nào đó.

1.1.3. Mối quan hệ giữa cấu trúc dữ liệu và giải thuật

Mối quan hệ giữa Cấu trúc dữ liệu và Giải thuật có thể minh họa bằng đẳng thức:

Cấu trúc dữ liệu + Giải thuật = Chương trình

Như vậy, khi đã có cấu trúc dữ liệu tốt, nắm vững giải thuật thực hiện thì việc thể hiện chương trình bằng một ngôn ngữ cụ thể chỉ là vấn đề thời gian. Khi có cấu trúc dữ liệu mà chưa tìm ra giải thuật thì không thể có chương trình và ngược lại không thể có giải thuật khi chưa có cấu trúc dữ liệu. Một chương trình máy tính chỉ có thể được hoàn thiện khi có đầy đủ cả Cấu trúc dữ liệu để lưu trữ dữ liệu lẫn Giải thuật xử lý dữ liệu theo yêu cầu của bài toán đặt ra.

1.2. Các tiêu chuẩn đánh giá cấu trúc dữ liệu và giải thuật

1.2.1. Các tiêu chuẩn đánh giá cấu trúc dữ liệu

Để đánh giá một cấu trúc dữ liệu chúng ta thường dựa vào một số tiêu chí sau:

- Cấu trúc dữ liệu phải tiết kiệm tài nguyên (bộ nhớ trong).
- Cấu trúc dữ liệu phải phản ảnh đúng thực tế của bài toán.
- Cấu trúc dữ liệu phải dễ dàng trong việc thao tác dữ liệu.

1.2.2. Đánh giá độ phức tạp của giải thuật

Việc đánh giá độ phức tạp của một giải thuật quả không dễ dàng chút nào. Ở đây, chúng ta chỉ muốn ước lượng thời gian thực hiện thuận toán T(n) để có thể có sự so sánh tương đối giữa các thuật toán với nhau. Trong thực tế, thời gian thực hiện một giải thuật còn phụ thuộc rất nhiều vào các điều kiện khác như cấu tạo của máy tính, dữ liệu đưa vào, … Ở đây chúng ta chỉ xem xét trên mức độ của lượng dữ liệu đưa vào ban đầu cho thuật toán thực hiện.

Để ước lượng thời gian thực hiện thuật toán chúng ta có thể xem xét thời gian thực hiện thuật toán trong hai trường hợp:

- Trong trường hợp tốt nhất: Tmin
- Trong trường hợp xấu nhất: Tmax

Từ đó chúng ta có thể ước lượng thời gian thực hiện trung bình của thuật toán: Tavg

2. TRỪU TƯỢNG HÓA DỮ LIỆU

2.1. Định nghĩa kiểu dữ liệu

Kiểu dữ liệu T có thể xem như là sự kết hợp của 2 thành phần:

- Miền giá trị mà kiểu dữ liệu T có thể lưu trữ: V
- Tập hợp các phép toán để thao tác dữ liệu: O

T = <V, O>

Mỗi kiểu dữ liệu thường được đại diện bởi một tên (định danh). Mỗi phần tử dữ liệu có kiểu T sẽ có giá trị trong miền V và có thể được thực hiện các phép toán thuộc tập hợp các phép toán trong O.

Để lưu trữ các phần tử dữ liệu này thường phải tốn một số byte(s) trong bộ nhớ, số byte(s) này gọi là kích thước của kiểu dữ liệu.

2.2. Các kiểu dữ liệu cơ sở

Hầu hết các ngôn ngữ lập trình đều có cung cấp các kiểu dữ liệu cơ sở. Tùy vào mỗi ngôn ngữ mà các kiểu dữ liệu cơ sở có thể có các tên gọi khác nhau song chung quy lại có những loại kiểu dữ liệu cơ sở như sau:

- Kiểu số nguyên: Có thể có dấu hoặc không có dấu và thường có các kích thước như sau:

+ Kiểu số nguyên 1 byte

+ Kiểu số nguyên 2 bytes

+ Kiểu số nguyên 4 bytes

Kiểu số nguyên thường được thực hiện với các phép toán:

O = {+, - , * , / , %, < , >, <= , >= , == , …}

- Kiểu số thực: Thường có các kích thước sau:

  + Kiểu số thực 4 bytes

  + Kiểu số thực 6 bytes

  + Kiểu số thực 8 bytes

  + Kiểu số thực 10 bytes

  Kiểu số thực thường được thực hiện với các phép toán:

  O = {+, - , * , / , < , > , <= , >=, == , …}

- Kiểu ký tự: Có thể có các kích thước sau:

  + Kiểu ký tự 1 byte

  + Kiểu ký tự 2 bytes

  Kiểu ký tự thường được thực hiện với các phép toán:

  O = { < , > , <= , >= , ==, isalpha, isdigit, …}

- Kiểu chuỗi ký tự: Có kích thước tùy thuộc vào từng ngôn ngữ lập trình.

  Kiểu chuỗi ký tự thường được thực hiện với các phép toán:

  O = { strcmp, strlen, strcat, strcpy …}

- Kiểu luận lý: Thường có kích thước 1 byte.

  Kiểu luận lý thường được thực hiện với các phép toán:

  O = {!, &&, ||, <, > , <= , >= , == , …}

2.3. Các kiểu dữ liệu có cấu trúc

Kiểu dữ liệu có cấu trúc là các kiểu dữ liệu được xây dựng trên cơ sở các kiểu dữ liệu đã có (có thể là một kiểu dữ liệu có cấu trúc khác). Tùy vào từng ngôn ngữ lập trình song thường có các loại sau:

- Kiểu mảng hay còn gọi là dãy: kích thước bằng tổng kích thước của các phần tử.
- Kiểu mẫu tin hay cấu trúc: kích thước bằng tổng kích thước các thành phần của nó (Field).

2.4. Một số kiểu dữ liệu khác

Các ngôn ngữ lập trình thường cung cấp cho chúng ta một kiểu dữ liệu đặc biệt để lưu trữ các địa chỉ của bộ nhớ, đó là con trỏ (Pointer). Tùy vào loại con trỏ gần

(near pointer) hay con trỏ xa (far pointer) mà kiểu dữ liệu con trỏ sẽ có các kích thước khác nhau:

- Con trỏ gần: 2 bytes
- Con trỏ xa: 4 bytes

Ngoài ra đa số các ngôn ngữ lập trình còn có kiểu dữ liệu tập tin. Tập tin (File) có thể xem là một kiểu dữ liệu đặc biệt, kích thước tối đa của tập tin tùy thuộc vào không gian đĩa nơi lưu trữ tập tin. Việc đọc, ghi dữ liệu trực tiếp trên tập tin rất mất thời gian và không bảo đảm an toàn cho dữ liệu trên tập tin đó. Do vậy, trong thực tế, chúng ta không thao tác trực tiếp dữ liệu trên tập tin mà chúng ta cần chuyển từng phần hoặc toàn bộ nội dung của tập tin vào bộ nhớ trong để xử lý.

3. ĐÁNH GIÁ ĐỘ PHỨC TẠP CỦA GIẢI THUẬT

3.1. Các bước phân tích giải thuật

Bước đầu tiên trong việc phân tích một giải thuật là xác định đặc trưng dữ liệu sẽ được dùng làm dữ liệu nhập của thuật toán và quyết định phân tích nào là thích hợp. Về mặt lý tưởng chúng ta muốn rằng với một phân bố tùy ý được cho của dữ liệu nhập, sẽ có sự phân bố tương ứng về thời gian hoạt động của thuật toán. Chúng ta không thể đạt đến điều lý tưởng này cho bất kỳ một thuật toán không tầm thường nào, vì vậy chúng ta chỉ quan tâm về tính năng của thuật toán bằng cách cố gắng chứng minh thời gian chạy luôn luôn nhỏ hơn một "chặn trên" bất chấp dữ liệu nhập như thế nào và cố gắng tính được thời gian chạy trung bình cho dữ liệu nhập "ngẫu nhiên".

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

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

Nói chung, tính năng của một giải thuật thường có thể được phân tích ở một mức độ vô cùng chính xác, chỉ bị giới hạn bởi tính năng không chắc chắn của máy tính hay bởi sự khó khăn trong việc xác định các tính chất toán học của một vài đại lượng trừu tượng. Tuy nhiên, thay vì phân tích một cách chi tiết chúng ta thường ước lượng để tránh sa vào chi tiết.

3.2. Sự phân lớp các giải thuật

Hầu hết các giải thuật đều có một tham số chính là N. Thông thường đó là số lượng các phần tử dữ liệu được xử lý mà ảnh hưởng rất nhiều tới thời gian thực thi chương trình. Tham số N có thể là bậc của một đa thức, kích thước của một tập tin được sắp xếp hay tìm kiếm, số nút trong một đồ thị,...

Đa số các thuật toán trong giáo trình này có thời gian thực thi tiệm cận tới một trong các hàm sau:

3.2.1. Lớp có hàm là hằng số

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

3.2.2. Lớp có hàm là LogN

Khi thời gian chạy của chương trình là *logarit*, tức là thời gian chạy chương trình tiến chậm khi N lớn dần. Thời gian chạy loại này xuất hiện trong các chương trình mà giải một bài toán lớn bằng cách chuyển nó thành các bài toán nhỏ hơn bằng cách cắt bỏ kích thước bớt một hằng số nào đó. Với mục đích của chúng ta, thời gian chạy có được xem như nhỏ hơn một hằng số "lớn". Cơ số của *logarit* làm thay đổi hằng số đó nhưng không nhiều, chẳng hạn khi N = 1000 thì logN là 3 nếu cơ số là 10 và là 10 nếu cơ số là 2; khi N = 1000000, logN được nhân gấp đôi. Bất cứ khi nào N được nhân gấp đôi, logN được tăng lên thêm một hằng số, nhưng logN không được nhân gấp đôi tới khi N tăng tới N^2.

3.2.3. Lớp có hàm là N

Khi thời gian chạy của chương trình là *tuyến tính*, nói chung đây là trường hợp mà một số lượng nhỏ các xử lý được làm cho mỗi phần tử dữ liệu nhập.

Khi N = 1000000 thì thời gian chạy cũng cỡ như vậy.

Khi N được nhân gấp đôi thì thời gian chạy cũng được nhân gấp đôi. Đây là tình huống tối ưu cho một thuật toán mà phải xử lý N dữ liệu nhập (hay sinh ra N dữ liệu xuất).

3.2.4. Lớp có hàm là NLogN

Đây là thời gian chạy tăng dần lên cho các thuật toán mà giải một bài toán bằng cách tách nó thành các bài toán con nhỏ hơn, kế đến giải quyết chúng một cách độc lập và sau đó tổng hợp các lời giải. Bởi vì thiếu một tính từ tốt hơn (có lẽ là "tuyến tính logarit" ?), chúng ta nói rằng thời gian chạy của thuật toán như thế là "NlogN".

Khi N = 1000000, NlogN có lẽ khoảng 6 triệu.

Khi N được nhân gấp đôi, thời gian chạy bị nhân lên nhiều hơn gấp đôi (nhưng không nhiều lắm).

3.2.5. Lớp có hàm là N^2

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

Khi N = 1000 thì thời gian chạy là 1000000.

Khi N được nhân đôi thì thời gian chạy tăng lên gấp 4 lần.

3.2.6. Lớp có hàm là N^3

Tương tự, một thuật toán mà xử lý một bộ 3 của các phần tử dữ liệu (có lẽ 3 vòng lặp lồng nhau) có thời gian chạy bậc 3 và cũng chỉ có ý nghĩa thực tế trong các bài toán nhỏ.

Khi N = 100 thì thời gian chạy là 1000000.

Khi N được nhân đôi thì thời gian chạy tăng lên gấp 8 lần.

3.2.7. Lớp có hàm là hàm mũ (2^N, N!, N^N)

Một số ít thuật toán có thời gian chạy lũy thừa lại thích hợp trong một số trường hợp thực tế, mặc dù các thuật toán như thế là "sự ép buộc thô bạo" để giải bài toán.

Khi N = 20 thì thời gian chạy xấp xỉ là 1000000.

Khi N được nhân đôi thì thời gian chạy được nâng lên lũy thừa 2.

Thời gian chạy của một chương trình cụ thể đôi khi là một hằng số nhân với các số hạng nói trên cộng thêm một số hạng nhỏ hơn. Các giá trị của hằng số và các số hạng phụ thuộc vào các kết quả của sự phân tích và các chi tiết cài đặt. Hệ số của hằng số liên quan đến số chỉ thị bên trong vòng lặp: ở một tầng tùy ý của thiết kế thuật toán thì phải cẩn thận giới hạn số chỉ thị như thế. Với N lớn thì các hằng số đóng vai trò chủ chốt, với N nhỏ thì các số hạng cũng đóng vai trò quan trọng và sự so sánh thuật toán sẽ khó khăn hơn. Ngoài những hàm vừa nói trên còn có một số hàm khác, ví dụ như một thuật toán với N^2 phần tử dữ liệu nhập mà có thời gian thực thi là bậc 3 theo N thì sẽ được phân lớp như một thuật toán $N^{3/2}$. Một số thuật toán có 2 giai đoạn phân tách thành các bài toán con và có thời gian chạy xấp xỉ với $N\log^2 N$.

3.3. Phân tích trường hợp trung bình

Một tiếp cận trong việc nghiên cứu tính năng của thuật toán là khảo sát trường hợp trung bình. Trong tình huống đơn giản nhất, chúng ta có thể đặc trưng chính xác các dữ liệu nhập của thuật toán. Ví dụ một thuật toán sắp xếp có thể thao tác trên một mảng N số nguyên ngẫu nhiên, hay một thuật toán hình học có thể xử lý N điểm ngẫu nhiên trên mặt phẳng với các tọa độ nằm giữa 0 và 1. Kế đến là tính toán thời gian thực hiện trung bình của mỗi chỉ thị, và tính thời gian chạy trung bình của chương trình bằng cách nhân tần số sử dụng của mỗi chỉ thị với thời gian cần cho chỉ thị đó, sau đó cộng tất cả chúng lại với nhau. Tuy nhiên có ít nhất 3 khó khăn trong các tiếp cận này như sau:

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

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

Thứ ba, trong việc phân tích trường hợp trung bình là mô hình dữ liệu nhập có thể không đặc trưng đầy đủ dữ liệu nhập mà chúng ta gặp trong thực tế. Ví dụ như làm thế nào để đặc trưng được dữ liệu nhập cho chương trình xử lý văn bản tiếng Anh? Một tác giả đề nghị nên dùng các mô hình dữ liệu nhập chẳng hạn như "tập tin thứ tự ngẫu nhiên" cho thuật toán sắp xếp, hay "tập hợp điểm ngẫu nhiên" cho thuật toán hình học, đối với những mô hình như thế thì có thể đạt được các kết quả toán học mà tiên đoán được tính năng của các chương trình chạy trên các ứng dụng thông thường.

BÀI TẬP CHƯƠNG 2

-----ଓ✧ଌ-----

1. Trình bày tầm quan trọng của Cấu trúc dữ liệu và Giải thuật đối với người lập trình?

2. Các tiêu chuẩn để đánh giá cấu trúc dữ liệu và giải thuật?

3. Khi xây dựng giải thuật có cần thiết phải quan tâm tới cấu trúc dữ liệu hay không? Tại sao?

4. Liệt kê các kiểu dữ liệu cơ sở, các kiểu dữ liệu có cấu trúc trong C, Pascal?

5. Sử dụng các kiểu dữ liệu cơ sở trong C, hãy xây dựng cấu trúc dữ liệu để lưu trữ trong bộ nhớ trong (RAM) của máy tính đa thức có bậc tự nhiên n ($0 \le n \le 100$) trên trường số thực (a_i , $x \in R$):

$$fn(x) = \sum_{i=1}^{n} a_i x^i$$

Với cấu trúc dữ liệu được xây dựng, hãy trình bày thuật toán và cài đặt chương trình để thực hiện các công việc sau:

a. Nhập các đa thức.

b. Xuất các đa thức.

c. Tính giá trị của đa thức tại giá trị x_0 nào đó.

d. Tính tổng của hai đa thức.

e. Tính tích của hai đa thức.

6. Tương tự như bài tập 5, nhưng đa thức trong trường số hữu tỷ Q (các hệ số a_i và x là các phân số có tử số và mẫu số là các số nguyên).

7. Sử dụng kiểu dữ liệu cấu trúc trong C, hãy xây dựng cấu trúc dữ liệu để lưu trữ trong bộ nhớ trong (RAM) của máy tính trạng thái của các cột đèn giao thông (có 3 đèn: Xanh, Vàng và Đỏ). Với cấu trúc dữ liệu đã được xây dựng, hãy trình bày thuật toán và cài đặt chương trình để mô phỏng (minh họa) cho hoạt động của 2 cột đèn trên hai tuyến đường giao nhau tại một ngã tư.

8. Sử dụng các kiểu dữ liệu cơ sở trong C, hãy xây dựng cấu trúc dữ liệu để lưu trữ trong bộ nhớ trong (RAM) của máy tính trạng thái của một bàn cờ CARO có kích thước M×N ($0 \le M, N \le 20$). Với cấu trúc dữ liệu được xây dựng, hãy trình bày thuật toán và cài đặt chương trình để thực hiện các công việc sau:

a. In ra màn hình bàn cờ CARO trong trạng thái hiện hành.

b. Kiểm tra xem có ai thắng hay không? Nếu có thì thông báo "Kết thúc", nếu không thì thông báo "Tiếp tục".

CHƯƠNG 3: DANH SÁCH

-----ଓ✧ଌ-----

1. KHÁI NIỆM

Mô hình toán học của danh sách là một tập hợp hữu hạn các phần tử có cùng một kiểu, mà tổng quát ta gọi là kiểu phần tử (elementtype). Ta biểu diễn danh sách như là một chuỗi các phần tử của nó: $a_1, a_2, \ldots, a_n$ với $n \geq 0$. Nếu $n = 0$ ta nói danh sách rỗng (empty list). Nếu $n > 0$ ta gọi a_1 là phần tử đầu tiên và a_n là phần tử cuối cùng của danh sách. Số phần tử của danh sách ta gọi là độ dài của danh sách.

Một tính chất quan trọng của danh sách là các phần tử của danh sách có thứ tự tuyến tính theo vị trí (position) xuất hiện của các phần tử. Ta nói a_i đứng trước a_{i+1}, với i từ 1 đến n-1, tương tự ta nói a_i là phần tử đứng sau a_{i-1}, với i từ 2 đến n. Ta cũng nói a_i là phần tử tại vị trí thứ i, hay phần tử thứ i của danh sách.

Ví dụ: Tập hợp họ tên các SV của lớp CNTTK37 được liệt kê trên giấy như sau:

1. Võ Thị Kim Dung
2. Lê Văn Huynh
3. Cao Thị Thùy Linh
4. Lâm Bích Phương
5. Lê Thị Lệ Sương

Đây chính là danh sách gồm có 5 phần tử, mỗi phần tử có một vị trí trong danh sách theo thứ tự xuất hiện của nó.

2. PHÂN LOẠI

Danh sách thường được phân thành 2 loại: danh sách đặc (cài đặt bằng mảng); danh sách liên kết (cài đặt bằng con trỏ). Ngoài ra còn có 2 loại danh sách đặc biệt được gọi là danh sách hạn chế đó là ngăn xếp (stack) và hàng đợi (queue). Tùy theo các phép toán thường được sử dụng trong danh sách mà người lập trình sẽ chọn cấu trúc dữ liệu thích hợp để cài đặt. Trong danh sách liên kết người ta lại chia thành các loại cụ thể hơn như: danh sách liên kết đơn, danh sách liên kết kép, danh sách liên kết vòng. Trong giáo trình này chúng ta sẽ nghiên cứu và cài đặt danh sách liên kết đơn.

3. CÁC PHÉP TOÁN CƠ BẢN

Để thiết lập kiểu dữ liệu trừu tượng danh sách (hay ngắn gọn là danh sách) ta phải định nghĩa các phép toán trên danh sách. Trên thực tế thì không có một tập hợp các phép toán nào là thích hợp cho mọi ứng dụng (application). Vì vậy ở đây ta sẽ định nghĩa một số phép toán cơ bản nhất trên danh sách. Để thuận tiện cho việc định nghĩa ta giả sử rằng danh sách gồm các phần tử có kiểu là kiểu phần tử (elementtype). Vị trí của các phần tử trong danh sách có kiểu là kiểu vị trí và vị trí sau phần tử cuối cùng trong danh sách L là ENDLIST(L) chính là NULL. Cần nhấn mạnh rằng khái niệm vị trí (position) là do ta định nghĩa, nó không phải là giá trị của các phần tử trong danh sách. Vị trí có thể là đồng nhất với vị trí lưu trữ phần tử hoặc không.

Các phép toán được định nghĩa trên danh sách là:

MakeNullList(L): Khởi tạo một danh sách L rỗng.

EmptyList(L): Cho kết quả TRUE nếu danh sách rỗng, ngược lại nó cho giá trị là FALSE.

InsertList(X, P, L): Xen phần tử X (kiểu elementtype) vào sau vị trí P (kiểu position) trong danh sách L. Tức là nếu danh sách là $a_1, a_2, \ldots, a_{p-1}, a_p, \ldots, a_n$ thì sau khi xen ta có kết quả $a_1, a_2, \ldots, a_{p-1}, a_p, X, \ldots, a_n$. Nếu vị trí P không tồn tại trong danh sách thì phép toán không được xác định.

First(L): Cho kết quả là vị trí của phần tử đầu tiên trong danh sách. Nếu danh sách rỗng thì ENDLIST(L) chính là NULL được trả về.

Last(L): Cho kết quả là vị trí của phần tử cuối cùng trong danh sách.

Next(P, L): Cho kết quả là vị trí của phần tử (kiểu position) đứng sau phần tử P; nếu P là phần tử cuối cùng trong danh sách L thì NEXT(P, L) cho kết quả là ENDLIST(L) chính là NULL. NEXT không xác định nếu P không phải là vị trí của một phần tử trong danh sách.

Previous(P, L): Cho kết quả là vị trí của phần tử đứng trước phần tử P trong danh sách. Nếu P là phần tử đầu tiên trong danh sách thì Previous(P, L) không xác định. Previous cũng không xác định trong trường hợp P không phải là vị trí của phần tử nào trong danh sách.

Locate(X, L): Thực hiện việc định vị phần tử có nội dung X xuất hiện đầu tiên trong danh sách L. Locate trả về kết quả là vị trí (kiểu position) của phần tử X trong danh sách. Nếu X không có trong danh sách thì vị trí sau phần tử cuối cùng của danh sách được trả về, tức là ENDLIST(L) chính là NULL.

Retrieve(P, L): Lấy giá trị của phần tử ở vị trí P (kiểu position) của danh sách L; nếu vị trí P không có trong danh sách thì kết quả không xác định (có thể báo lỗi).

DeleteList(P, L): Xóa phần tử ở sau vị trí P (kiểu position) của danh sách. Nếu vị trí P không có trong danh sách thì phép toán không được định nghĩa và danh sách L sẽ không thay đổi.

Trong thiết kế các giải thuật sau này chúng ta dùng các phép toán trừu tượng đã được định nghĩa ở đây như là các phép toán nguyên thủy.

Ví dụ: Dùng các phép toán trừu tượng trên danh sách, viết một hàm nhận một tham số là danh sách rồi sắp xếp danh sách theo thứ tự tăng dần (giả sử các phần tử trong danh sách thuộc kiểu có thứ tự).

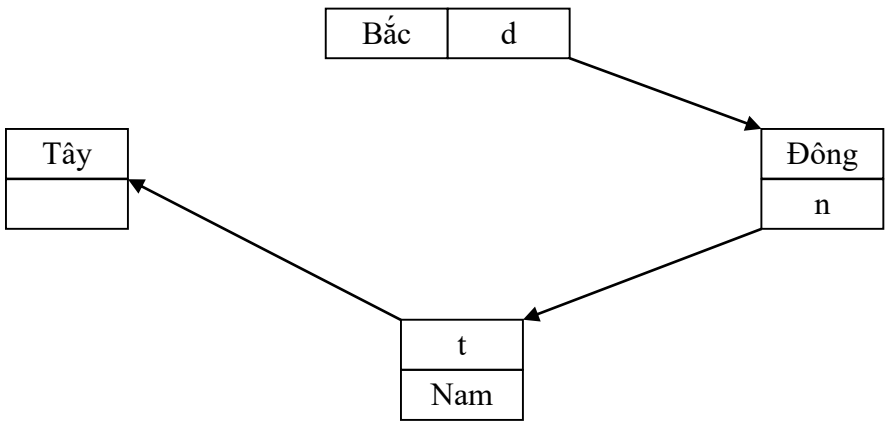
Giả sử hàm Swap(p, q) thực hiện việc hoán đổi giá trị của hai phần tử tại vị trí p và q trong danh sách, hàm sắp xếp được viết như sau:

```
void Sort(List &L)
{    Position p, q; //kiểu vị trí của các p.tử trong ds
     p = First(L); //vị trí phần tử đầu tiên trong ds
     while(p != NULL)
          {    q = Next(p,L);
               //vị trí phần tử đứng ngay sau phần tử p
               while(q != NULL)
                    {
                      if(Retrieve(p,L) > Retrieve(q,L))
                         Swap(p,q);
                         // dịch chuyển nội dung phần tử
                      q = Next(q,L);
                    }
               p = Next(p,L);
          }
}
```

Tuy nhiên, cần phải nhấn mạnh rằng, đây là các phép toán trừu tượng do chúng ta định nghĩa, nó chưa được cài đặt trong các ngôn ngữ lập trình. Do đó để cài đặt giải thuật thành một chương trình chạy được thì ta cũng phải cài đặt các phép toán thành các hàm trong chương trình. Chẳng hạn như ta phải cài đặt các hàm Swap, First, Next, Retrieve, …

4. CÀI ĐẶT

Có nhiều cách cài đặt danh sách khác nhau, trong đó có 2 cách thường được sử dụng nhất là sử dụng mảng (danh sách đặc) và sử dụng con trỏ (danh sách liên kết). Trong giáo trình này, chúng ta chọn cách dùng con trỏ để liên kết các ô nhớ chứa các phần tử. Với cách cài đặt này các phần tử của danh sách được lưu trữ trong các ô nhớ, mỗi ô có thể chỉ đến ô chứa phần tử kế tiếp trong danh sách, và danh sách như thế được gọi là ***danh sách liên kết***.

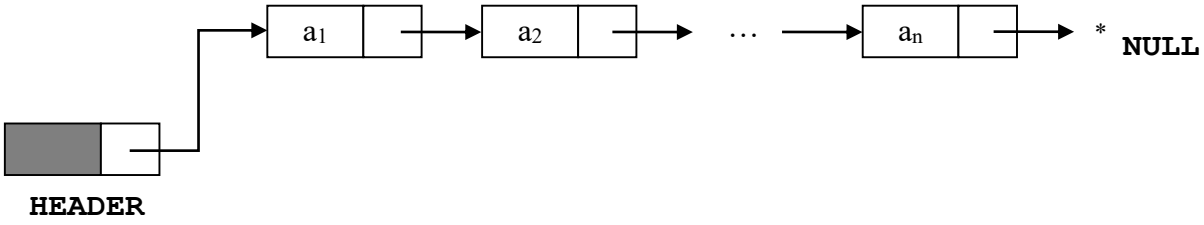


Ví dụ: Giả sử 1 lớp có 4 bạn: Đông, Tây, Nam, Bắc có địa chỉ lần lượt là d, t, n, b. Giả sử Đông có địa chỉ của Nam, Tây không có địa chỉ của bạn nào, Bắc giữ địa chỉ của Đông, Nam có địa chỉ của Tây được minh họa như hình trên.

Như vậy, nếu ta xét thứ tự các phần tử bằng cơ chế chỉ tới (trỏ đến) này thì ta có một danh sách: Bắc, Đông, Nam, Tây. Hơn nữa để có danh sách này thì ta cần và chỉ cần giữ địa chỉ của Bắc.

Trong cài đặt, để một ô có thể *chỉ đến* ô khác ta cài đặt mỗi ô là một mẫu tin (record, struct) có hai trường: trường Element chứa nội dung của các phần tử trong danh sách, trường Next là một *con trỏ* giữ địa chỉ của ô kế tiếp. Trường Next của phần tử cuối trong danh sách chỉ đến một giá trị đặc biệt là **NULL**.

Cấu trúc như vậy gọi là danh sách cài đặt bằng con trỏ hay *danh sách liên kết đơn* hay ngắn gọn là *danh sách liên kết*.



Để quản lý danh sách ta chỉ cần một biến giữ địa chỉ ô chứa phần tử đầu tiên của danh sách, tức là một con trỏ trỏ đến phần tử đầu tiên trong danh sách. Biến này gọi là *chỉ điểm đầu danh sách (Header)*. Header là một biến cùng kiểu với các ô chứa các phần tử của danh sách, tuy nhiên nó là một ô đặc biệt nên không chứa nội dung của bất kỳ phần tử nào của danh sách, trường Element là rỗng, trường Next trỏ tới ô chứa phần tử đầu tiên thật sự của danh sách. Nếu danh sách rỗng thì Header trỏ tới **NULL**.

Ở đây ta cần phân biệt rõ nội dung của một phần tử và địa chỉ của nó được giữ bởi phần tử nào trong cấu trúc trên. Ví dụ nội dung của phần tử đầu tiên trong danh sách trên là a_1, trong khi địa chỉ của nó được giữ bởi trường Next của Header. Tương tự các phần tử khác được cho trong bảng sau:

Phần tử thứ	Nội dung	Địa chỉ được giữ bởi
1	a_1	HEADER
2	a_2	1
3	a_3	2
...	...	...
n	a_n	(n-1)
Sau phần tử cuối cùng	Không xác định	n và n -> Next có giá trị là **NULL**

Như đã thấy trong bảng trên, địa chỉ của phần tử thứ i được giữ bởi phần tử thứ (i-1), như vậy để biết được địa chỉ của phần tử thứ i ta phải truy xuất vào ô thứ (i-1). Khi thêm hoặc xóa một phần tử trong danh sách liên kết tại vị trí p, ta phải cập nhật lại con trỏ trỏ tới vị trí này, tức là cập nhật lại (p-1). Nói cách khác, để thao tác vào địa chỉ p ta phải biết con trỏ trỏ vào p mà con trỏ này chính là (p-1).

Có thể nói nôm na rằng địa chỉ của phần tử a_i được giữ bởi ô đứng ngay phía trước ô chứa a_i. Hay nói chính xác hơn, địa chỉ của phần tử thứ i được giữ bởi con trỏ mà trường Next của nó trỏ tới ô chứa phần tử a_i. Như vậy địa chỉ của phần tử thứ 1 được giữ bởi trường Next của Header, địa chỉ của phần tử thứ 2 được giữ bởi trường Next của phần tử a_1, địa chỉ của phần tử thứ 3 được giữ bởi trường Next của phần tử a_2, ..., địa chỉ phần tử thứ n được giữ bởi trường Next của phần tử a_{n-1}. Và trường Next của phần tử a_n (sau cùng) sẽ chỉ tới NULL.

4.1. Khai báo cấu trúc dữ liệu

```
typedef <Kiểu_dữ_liệu> ElementType;
struct Node
{
     ElementType Element;
     Node* Next;
};
typedef Node* Position;
typedef Node* List;
```

4.2. Tạo danh sách rỗng

Như đã nói ở phần trên, ta dùng Header như là một biến con trỏ có kiểu giống như kiểu của một ô chứa một phần tử của danh sách. Tuy nhiên trường Element của Header không bao giờ được dùng, chỉ có trường Next dùng để trỏ tới ô chứa phần tử đầu tiên của danh sách. Vậy nếu như danh sách rỗng thì ô Header vẫn phải tồn tại và ô này có trường Next chỉ đến **NULL** (do không có phần tử nào). Vì vậy khi khởi tạo danh sách rỗng, ta phải cấp phát ô nhớ cho Header và cho con trỏ trong trường Next của nó trỏ tới **NULL**.

```
void MakeNullList(List &L)
{
    L=new Node;
    L->Next=NULL;
}
```

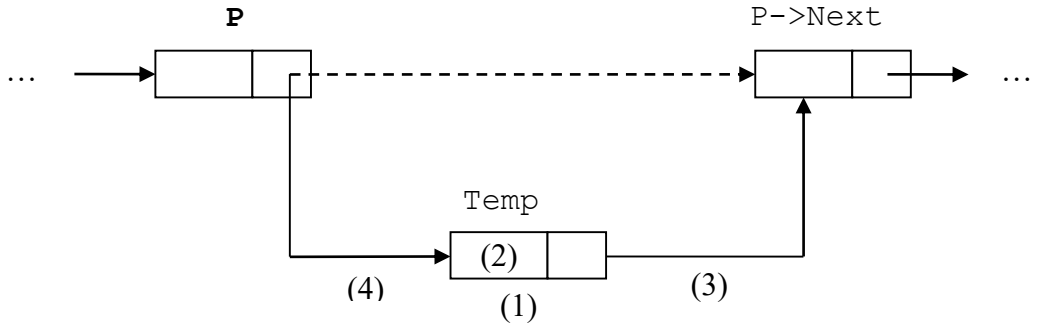
4.3. Kiểm tra danh sách rỗng

Danh sách rỗng nếu như trường Next trong ô Header trỏ tới **NULL**.

```
bool EmptyList(List L)
{
    return L->Next==NULL;
}
```

4.4. Thêm một phần tử

Thêm (Xen/Chèn) một phần tử có giá trị X vào danh sách L ngay sau vị trí P ta phải cấp phát một ô nhớ để lưu trữ phần tử mới này và nối kết lại các con trỏ để đưa ô mới này vào vị trí P. Sơ đồ nối kết và thứ tự các thao tác được minh họa như sau:




```
void InsertList(ElementType X, Position P, List &L)
{
     Position Temp;
     Temp=new Node;
     Temp->Element=X;
     Temp->Next=P->Next;
     P->Next=Temp;
}
```

4.5. Xác định phần tử đầu tiên

```
Position First(List L)
{
    Position P=L->Next;
    return P;
}
```

4.6. Xác định phần tử cuối cùng

```
Position Last(List L)
{
    Position P=L;
```

```
    while(P->Next!=NULL)
        P=P->Next;
    return P;
}
```

4.7. Xác định phần tử đứng ngay sau P

```
Position Next(Position P, List L)
{
    return P->Next;
}
```

4.8. Xác định phần tử đứng ngay trước P

```
Position Previous(Position P, List L)
{
    Position Temp;
    if(P==L) return NULL;
    else
    {
        Temp=L;
        while(Temp->Next!=P) Temp=Temp->Next;
        return Temp;
    }
}
```

4.9. Định vị một phần tử

Để định vị phần tử X trong danh sách L ta tiến hành tìm từ đầu danh sách (ô Header) nếu tìm thấy thì vị trí của phần tử đầu tiên được tìm thấy sẽ được trả về nếu không thì ENDLIST(L) chính là NULL được trả về. Nếu X có trong danh sách thì hàm Locate trả về vị trí P mà trong đó ta có X = P -> Element.

```
Position Locate(ElementType X, List L)
{
     int found=0;
     Position P=L;
     while(P->Next!=NULL && found==0)
         if(P->Next->Element==X)
             found=1;
```

```
        else
            P=P->Next;
    return P->Next;
}
```

Thực chất, khi gọi hàm Locate ở trên ta có thể truyền giá trị cho L là bất kỳ giá trị nào. Nếu L là Header thì hàm sẽ tìm X từ đầu danh sách. Nếu L là một vị trí P bất kỳ trong danh sách thì hàm Locate sẽ tiến hành định vị phần tử X từ vị trí P.

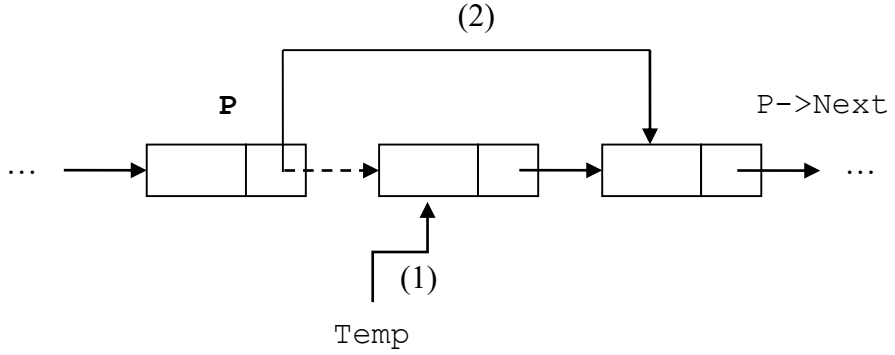
4.10. Xác định nội dung một phần tử

Nội dung phần tử đang lưu trữ tại vị trí P trong danh sách L là P -> Element. Do đó, hàm sẽ trả về giá trị P -> Element nếu phần tử có tồn tại, ngược lại phần tử không tồn tại (P = **NULL**) thì hàm không xác định và trả về giá trị là 0 (Trong L không có phần tử nào có giá trị là 0).

```
ElementType Retrieve(Position P, List L)
{
    if(P!=NULL)
        return P->Element;
    else
        return 0;
}
```

4.11. Xóa phần tử ngay sau P

Tương tự như khi xen một phần tử vào danh sách liên kết, muốn xóa một phần tử khỏi danh sách ta cần biết vị trí P của phần tử ngay trước phần tử muốn xóa trong danh sách L. Nối kết lại các con trỏ bằng cách cho P trỏ tới phần tử đứng ngay sau phần tử muốn xóa. Trong các ngôn ngữ lập trình không có cơ chế thu hồi vùng nhớ tự động như ngôn ngữ Pascal, C thì ta phải thu hồi vùng nhớ của ô bị xóa một các tường minh trong giải thuật. Tuy nhiên vì tính đơn giản của giải thuật cho nên đôi khi chúng ta không đề cập đến việc thu hồi vùng nhớ cho các ô bị xóa. Chi tiết và trình tự các thao tác để xóa một phần tử trong danh sách liên kết như sau:



```
void DeleteList(Position P, List &L)
{    Position Temp;
     if(P->Next!=NULL)
          {
               Temp=P->Next;
               P->Next=Temp->Next;
               delete Temp;
          }
}
```

Để có thể thực thi được các hàm trên ta cần xây dựng thêm các hàm dùng để nhập và in danh sách. Và sau cùng là ta viết hàm main() để gọi thực hiện chúng.

4.12. Nhập danh sách

```
void ReadList(List &L)
{    ElementType X;
     do{
          cout<<"Nhap noi dung phan tu: ";
          cin>>X;
          if(X!=0)InsertList(X, Last(L),L);
      }while(X!=0);
}
```

4.13. In danh sách

```
void PrintList(List L)
{
    if(EmptyList(L)!=0)
        cout<<"Danh sach rong"<<endl;
    else
        while(L->Next!=NULL)
        {
               cout<<L->Next->Element<<'\t';
               L=L->Next;
         }
}
```

4.14. Hàm main()

```
main()
{
     List L;
     Position P;
     ElementType X,ND;
     MakeNullList(L);
     cout<<"=====NHAP DANH SACH LIEN KET====="<<endl;
     ReadList(L);
     cout<<"=====IN DANH SACH LIEN KET====="<<endl;
     PrintList(L);
     cout<<"Phan tu dau tien trong danh sach la: ";
     cout<<First(L)->Element<<endl;
     cout<<"Phan tu cuoi cung trong danh sach la: ";
     cout<<Last(L)->Element<<endl;
     cout<<"Nhap noi dung phan tu can them: ";
     cin>>X;
     cout<<"Ban muon them sau phan tu nao: ";
     cin>>ND;
     P=Locate(ND,L);
     if(P==NULL) cout<<"Khong them duoc"<<endl;
     else InsertList(X,P,L);
     cout<<"Ban muon them truoc phan tu nao: ";
     cin>>ND;
     P=Locate(ND,L);
     P=Previous(P,L);
     if(P==NULL) cout<<"Khong them duoc"<<endl;
     else InsertList(X,P,L);
     cout<<"===DANH SACH LIEN KET SAU KHI THEM==="<<endl;
     PrintList(L);
     cout<<"Ban muon xoa phan tu sau phan tu nao: ";
     cin>>ND;
```

```
    P=Locate(ND,L);
    if(P==NULL) cout<<"Khong xoa duoc"<<endl;
    else DeleteList(P,L);
    cout<<"===DANH SACH LIEN KET SAU KHI XOA==="<<endl;
    PrintList(L);
}
```

5. NGĂN XẾP (STACK)

5.1. Định nghĩa

Ngăn xếp (Stack) là một danh sách mà ta giới hạn việc thêm vào hoặc loại bỏ một phần tử chỉ thực hiện tại một đầu của danh sách, đầu này gọi là đỉnh (TOP) của ngăn xếp.

Ta có thể xem hình ảnh trực quan của ngăn xếp bằng một chồng đĩa đặt trên bàn. Muốn thêm vào chồng đó 1 đĩa ta để đĩa mới trên đỉnh chồng, muốn lấy các đĩa ra khỏi chồng ta cũng phải lấy đĩa trên trước. Như vậy ngăn xếp là một cấu trúc có tính chất "vào sau - ra trước" hay "vào trước - ra sau" (**LIFO** (last in - first out) hay **FILO** (first in - last out)).

5.2. Các phép toán cơ bản

MAKENULL_STACK(S): Tạo một ngăn xếp rỗng.

EMPTY_STACK(S): Kiểm tra ngăn xếp rỗng. Hàm cho kết quả 1 (true) nếu ngăn xếp rỗng và 0 (false) trong trường hợp ngược lại.

PUSH(x, S): Thêm một phần tử x vào đỉnh ngăn xếp.

TOP(S): Trả về giá trị của phần tử tại đỉnh ngăn xếp.

POP(S): Xóa phần tử tại đỉnh ngăn xếp.

Như đã nói từ trước, khi thiết kế giải thuật ta có thể dùng các phép toán trừu tượng như là các "nguyên thủy" mà không cần phải định nghĩa lại hay giải thích thêm. Tuy nhiên để giải thuật đó thành chương trình chạy được thì ta phải chọn một cấu trúc dữ liệu hợp lý để cài đặt các "nguyên thủy" này.

Ví dụ: Viết Reverse nhận một chuỗi kí tự từ bàn phím cho đến khi gặp ký tự @ thì kết thúc việc nhập và in kết quả theo thứ tự ngược lại.

```
void Reverse()
{
    Stack S;
    char c;
    MakeNull_Stack(S);
```

```
    do{
        cin>>c;
        Push(c,S);
    }while(c!='@');
    cout<<"Chuoi theo thu tu nguoc lai la"<<endl;
    while(!Empty_Stack(S))
    {
        cout<<Top(S);
        Pop(S);
    }
}
```

5.3. Cài đặt

Ngăn xếp là một danh sách đặc biệt nên ta có thể sử dụng kiểu dữ liệu trừu tượng danh sách để biểu diễn cách cài đặt nó. Như vậy, ta có thể khai báo ngăn xếp tương tự như danh sách liên kết.

5.3.1. Khai báo cấu trúc dữ liệu

```
typedef <Kiểu_dữ_liệu> ElementType;
struct Node
{
    ElementType Element;
    Node* Next;
};
typedef Node* Position;
typedef Node* Stack;
```

Khi chúng ta đã dùng danh sách để biểu diễn cho ngăn xếp thì ta nên sử dụng các phép toán trên danh sách để cài đặt các phép toán trên ngăn xếp. Sau đây là phần cài đặt ngăn xếp bằng danh sách liên kết.

5.3.2. Tạo ngăn xếp rỗng

```
void MakeNullStack(Stack &S)
{
    S=NULL;
}
```

5.3.3. Kiểm tra ngăn xếp rỗng

```
bool EmptyStack(Stack S)
{
     if(S==NULL) return true;
     else return false;
}
```

5.3.4. Thêm một phần tử

```
void Push(ElementType X, Stack &S)
{
     Position Temp;
     Temp=new Node;
     Temp->Element=X;
     Temp->Next=S;
     S=Temp;
}
```

5.3.5. Trả về giá trị của phần tử ở đỉnh ngăn xếp

```
ElementType Top(Stack S)
{
     if(EmptyStack(S))
          return 0;
     else
          return S->Element;
}
```

5.3.6. Xóa một phần tử

```
void Pop(Stack &S)
{
     if(!EmptyStack(S)){
          Position Temp=S;
          S=S->Next;
          delete Temp;
     }
}
```

5.3.7. Nhập ngăn xếp

```
void ReadStack(Stack &S)
{    ElementType X;
     do{
         cout<<"Nhap noi dung phan tu: ";
         cin>>X;
         if(X!=0)Push(X,S);
    }while(X!=0);
}
```

5.3.8. In và xóa ngăn xếp

```
void PrintStack(Stack &S)
{
    if(EmptyStack(S))
         cout<<"Ngan xep rong"<<endl;
    else
         while(S!=NULL){
               cout<<Top(S)<<'\t';
               Pop(S);
         }
}
```

5.3.9. Hàm main( )

```
main()
{
        Stack S;
        MakeNullStack(S);
        cout<<"=====NHAP NGAN XEP====="<<endl;
        ReadStack(S);
        cout<<"=====IN NGAN XEP====="<<endl;
        PrintStack(S);
        cout<<endl;
}
```

5.4 Ứng dụng

Nếu một hàm đệ qui P(x) được gọi từ chương trình chính ta nói hàm được thực hiện ở mức 1. Hàm này gọi chính nó, ta nói nó đi sâu vào mức 2, ... cho đến một mức k. Rõ ràng mức k phải thực hiện xong thì mức k-1 mới được thực hiện tiếp tục, hay ta còn nói là hàm quay về mức k-1.

Trong khi một hàm từ mức i đi vào mức i+1 thì các biến cục bộ của mức i và địa chỉ của mã lệnh còn dang dở phải được lưu trữ, địa chỉ này gọi là địa chỉ trở về. Khi từ mức i+1 quay về mức i các giá trị đó được sử dụng. Như vậy những biến cục bộ và địa chỉ lưu sau được dùng trước. Tính chất này gợi ý cho ta dùng một ngăn xếp để lưu giữ các giá trị cần thiết của mỗi lần gọi tới hàm. Mỗi khi lùi về một mức thì các giá trị này được lấy ra để tiếp tục thực hiện mức này.

Ta có thể tóm tắt quá trình như sau:

Bước 1: Lưu các biến cục bộ và địa chỉ trở về.

Bước 2: Nếu thỏa điều kiện ngừng đệ qui thì chuyển sang bước 3. Nếu không thì tính toán từng phần và quay lại bước 1 (đệ qui tiếp).

Bước 3: Khôi phục lại các biến cục bộ và địa chỉ trở về.

Ví dụ sau đây minh họa việc dùng ngăn xếp để loại bỏ chương trình đệ qui của bài toán xuất ngược một danh sách liên kết.

```
#include <iostream>
using namespace std;
typedef int ElementType;
typedef struct Node
{
     ElementType Element;
     Node* Next;
};
typedef Node* Position;
typedef Node* List;
typedef Node* Stack;

void MakeNullList(List &L)
{
    L = new Node;
    L->Next=NULL;
}
```

```
bool EmptyList(List L)
{
    return L->Next==NULL;
}
void InsertList(ElementType X, Position P, List &L)
{    Position Temp;
     Temp=new Node;
     Temp->Element=X;
     Temp->Next=P->Next;
     P->Next=Temp;
}
Position Last(List L)
{
    Position P=L;
    while(P->Next!=NULL)
        P=P->Next;
    return P;
}
void ReadList(List &L)
{   ElementType X;
    do{
        cout<<"Nhap noi dung phan tu: ";
        cin>>X;
        if(X!=0)
            InsertList(X, Last(L),L);
    }while(X!=0);
}
void PrintListRecursion(List L)
{
     if(L->Next!=NULL)
    {
        cout<<L->Next->Element<<'\t';
        PrintListRecursion(L->Next);
    }
}
```

```
void PrintList(List L)
{
    if(EmptyList(L)!=0)
         cout<<"Danh sach rong"<<endl;
    else
         while(L->Next!=NULL)
         {
             cout<<L->Next->Element<<'\t';
             L=L->Next;
         }
}
void MakeNullStack(Stack &S)
{
    S=NULL;
}
bool EmptyStack(Stack S)
{
     if(S==NULL)
          return true;
     else
          return false;
}
void Push(ElementType X, Stack &S)
{    Position Temp;
     Temp=new Node;
     Temp->Element=X;
     Temp->Next=S;
     S=Temp;
}
ElementType Top(Stack S)
{
     if(EmptyStack(S))
          return 0;
     else
          return S->Element;
}
```

```
void Pop(Stack &S)
{
     if(!EmptyStack(S))
     {
          Position Temp=S;
          S=S->Next;
          delete Temp;
     }
}
//Xuat nguoc de qui
void ContPrintListRecursion(List L)
{
    if(L->Next!=NULL)
    {
        ContPrintListRecursion(L->Next);
        cout<<L->Next->Element<<'\t';
    }
}
//Xuat nguoc khong de qui
void ContPrintList(List L)
{   Stack S;
    MakeNullStack(S);
    while(L->Next!=NULL){
        Push(L->Next->Element,S);
        L=L->Next;
    }
    while(!EmptyStack(S))
    {
         cout<<Top(S)<<'\t';
         Pop(S);
     }
}
main()
{    List L;
     MakeNullList(L);
```

```
    cout<<"=====NHAP DANH SACH LIEN KET====="<<endl;
    ReadList(L);
    cout<<"=====IN XUOI KHONG DE QUI====="<<endl;
    PrintList(L);
    cout<<endl;
    cout<<"=====IN XUOI DE QUI====="<<endl;
    PrintListRecursion(L);
    cout<<endl;
    cout<<"=====IN NGUOC DE QUI====="<<endl;
    ContPrintListRecursion(L);
    cout<<endl;
    cout<<"=====IN NGUOC KHONG DE QUI====="<<endl;
    ContPrintList(L);
}
```

6. HÀNG ĐỢI (QUEUE)

6.1. Định nghĩa

Hàng đợi, hay ngắn gọn là hàng (queue) cũng là một danh sách đặc biệt mà phép thêm vào chỉ thực hiện tại một đầu của danh sách, gọi là cuối hàng (REAR), còn phép loại bỏ thì thực hiện ở đầu kia của danh sách, gọi là đầu hàng (FRONT).

Xếp hàng mua vé xem phim là một hình ảnh trực quan của khái niệm trên, người mới đến thêm vào cuối hàng còn người ở đầu hàng mua vé và ra khỏi hàng, vì vậy hàng còn được gọi là cấu trúc "vào trước - ra trước" hay "vào sau - ra sau" (**FIFO** (first in - first out) hay **LILO** (last in - last out)).

6.2. Các phép toán cơ bản

MAKENULL_QUEUE(Q): Khởi tạo một hàng rỗng.

ENQUEUE(x, Q): Thêm phần tử x vào cuối hàng Q.

DEQUEUE(Q): Xóa phần tử tại đầu của hàng Q.

EMPTY_QUEUE(Q): Hàm kiểm tra hàng rỗng.

6.3. Cài đặt

Như đã trình bày trong phần ngăn xếp, ta hoàn toàn có thể dùng danh sách để biểu diễn cho một hàng và dùng các phép toán đã được cài đặt của danh sách để cài đặt các phép toán trên hàng. Tuy nhiên làm như vậy có khi sẽ không hiệu quả, chẳng hạn dùng danh sách cài đặt bằng mảng ta thấy lời gọi INSERT_LIST(x, ENDLIST(Q), Q) tốn một hằng thời gian trong khi lời gọi DELETE_LIST(FIRST(Q), Q) để xóa phần tử đầu hàng (phần tử ở vị trí 0 của mảng) ta phải tốn thời gian tỉ lệ với số các

phần tử trong hàng để thực hiện việc dời toàn bộ hàng lên một vị trí. Để cài đặt hiệu quả hơn ta phải có một suy nghĩ khác dựa trên tính chất đặc biệt của phép thêm và loại bỏ một phần tử trong hàng.

Tương tự như ngăn xếp, chúng ta cũng có thể cài đặt hàng đợi bằng nhiều cấu trúc dữ liệu khác nhau. Ở đây ta sử dụng con trỏ để minh họa cài đặt cho hàng. Cách tự nhiên nhất là dùng hai con trỏ front và rear để giữ địa chỉ phần tử đầu và cuối hàng. Hàng được cài đặt như một danh sách liên kết nhưng không sử dụng chỉ điểm đầu.

6.3.1. Khai báo cấu trúc dữ liệu

```
typedef <Kiểu_dữ_liệu> ElementType;
struct Node
{
     ElementType Element;
     Node* Next;
};
typedef Node* Position;
struct Queue
{
     Position Front, Rear;
};
```

6.3.2. Khởi tạo hàng đợi rỗng

Khi hàng rỗng Front và Rear cùng là NULL.

```
void MakeNullQueue(Queue &Q)
{
     Q.Front=NULL;
     Q.Rear=NULL;
}
```

6.3.3. Kiểm tra hàng đợi rỗng

Hàng rỗng nếu Front và Rear cùng là NULL.

```
bool EmptyQueue(Queue Q)
{
     if(Q.Front==NULL && Q.Rear==NULL) return true;
     else return false;
}
```

6.3.4. Thêm một phần tử

Thêm một phần tử vào hàng ta thêm vào sau Rear, sau đó cho Rear giữ địa chỉ của phần tử mới này. Ta lưu ý là phải xét hàng có rỗng hay không trước khi thêm để có cách xử lý khác nhau.

```
void EnQueue(ElementType X, Queue &Q)
{
     if(EmptyQueue(Q)){
          Q.Rear=new Node;
          Q.Front=Q.Rear;
     }
     else{
          Q.Rear->Next=new Node;
          Q.Rear=Q.Rear->Next;
     }
     Q.Rear->Element=X;
     Q.Rear->Next=NULL;
}
```

6.3.5. Trả về giá trị của phần tử ở đầu hàng đợi

```
ElementType FirstQueue(Queue Q)
{
     if(EmptyQueue(Q)) return 0;
     else return Q.Front->Element;
}
```

6.3.6 Xóa một phần tử

```
void DeQueue(Queue &Q)
{
     if(!EmptyQueue(Q)){
          Position T=Q.Front;
          Q.Front=Q.Front->Next;
          if(Q.Front==NULL) Q.Rear=NULL;
          delete T;
     }
}
```

6.3.7. Nhập hàng đợi

```
void ReadQueue(Queue &Q)
{
    ElementType X;
    do{
        cout<<"Nhap noi dung phan tu: ";
        cin>>X;
        if(X!=0)EnQueue(X,Q);
    }while(X!=0);
}
```

6.3.8. In và xóa hàng đợi

```
void PrintQueue(Queue &Q)
{
    if(EmptyQueue(Q))
        cout<<"Hang rong"<<endl;
    else
        while(!EmptyQueue(Q))
         {
             cout<<FirstQueue(Q)<<'\t';
             DeQueue(Q);
        }
}
```

6.3.9. Hàm main( )

```
main()
{
    Queue Q;
    MakeNullQueue(Q);
    cout<<"=====NHAP HANG DOI====="<<endl;
    ReadQueue(Q);
    cout<<"=====IN HANG DOI====="<<endl;
    PrintQueue(Q);
    cout<<endl;
}
```

6.4. Ứng dụng

Hàng đợi là một cấu trúc dữ liệu được dùng khá phổ biến trong thiết kế giải thuật. Bất kỳ nơi nào ta cần quản lý dữ liệu, quá trình, ... theo kiểu vào trước - ra trước đều có thể ứng dụng hàng đợi.

Ví dụ rất dễ thấy là quản lý in trên mạng, nhiều máy tính yêu cầu in đồng thời và ngay cả một máy tính cũng yêu cầu in nhiều lần. Nói chung có nhiều yêu cầu in dữ liệu, nhưng máy in không thể đáp ứng tức thời tất cả các yêu cầu đó nên chương trình quản lý in sẽ thiết lập một hàng đợi để quản lý các yêu cầu. Yêu cầu nào mà chương trình quản lý in nhận trước nó sẽ giải quyết trước.

Một ví dụ khác là duyệt cây theo mức, các giải thuật duyệt theo chiều rộng một đồ thị có hướng hoặc vô hướng cũng dùng hàng đợi để quản lý các nút đồ thị. Các giải thuật đổi biểu thức trung tố thành hậu tố, tiền tố.

BÀI TẬP CHƯƠNG 3

-----ଓ✧ଌ-----

1. Xây dựng các hàm sau trên danh sách liên kết, nội dung các phần tử kiểu số nguyên:

a. Thêm 1 phần tử vào sau Y xuất hiện sau cùng.

b. Thêm 1 phần tử vào trước Y xuất hiện ở lần thứ K.

c. Xóa phần tử X xuất hiện ở lần thứ K.

d. Xóa tất cả các phần tử X có trong danh sách.

e. In từ đầu danh sách đến X xuất hiện sau cùng.

f. In từ X xuất hiện ở lần thứ K đến cuối danh sách.

g. Tách 1 danh sách thành 2 danh sách tương ứng với nội dung chẵn và lẻ.

h. Ghép danh sách 2 vào chính giữa danh sách 1.

2. Xây dựng các hàm sau trên danh sách liên kết, nội dung các phần tử kiểu số thực:

a. Sắp xếp danh sách có thứ tự tăng dần.

b. Trộn 2 danh sách có thứ tự giảm dần thành 1 danh sách có thứ tự giảm dần.

c. Trộn 2 danh sách có thứ tự tăng dần thành 1 danh sách có thứ tự giảm dần.

d. Cho biết nội dung của phần tử nhỏ nhất.

e. Trả về danh sách bắt đầu là phần tử lớn nhất đầu tiên.

f. In từ phần tử lớn nhất đầu tiên đến phần tử nhỏ nhất sau cùng.

g. Cho biết phần tử có nội dung nhỏ nhất là phần tử thứ mấy trong ds.

h. Đếm số phần tử có phần nguyên là số nguyên tố.

i. Tính tổng các phần tử có phần nguyên là bội số của K.

3. Viết chương trình tạo một danh sách liên kết theo giải thuật thêm vào đầu danh sách, mỗi nút chứa một số nguyên.

4. Viết hàm tên Delete_Node để xóa nút có địa chỉ p. Viết một hàm loại bỏ tất cả các nút có nội dung x trong danh sách liên kết First.

5. Viết hàm Copy_List trên danh sách liên kết để tạo ra một danh sách liên kết mới giống danh sách liên kết cũ.

6. Viết hàm lọc danh sách liên kết để tránh trường hợp các nút trong danh sách liên kết bị trùng Element.

7. Viết hàm đảo ngược vùng liên kết của một danh sách liên kết sao cho:

- Header sẽ chỉ đến phần tử cuối
- Phần tử đầu có trường Next là NULL.

8. Viết hàm Left_Traverse để duyệt ngược danh sách liên kết.

9. Viết hàm tách một danh sách liên kết thành hai danh sách liên kết, trong đó một danh sách liên kết chứa các phần tử có số thứ tự lẻ và một danh sách liên kết chứa các phần tử có số thứ tự chẵn trong danh sách liên kết cũ.

10. Xây dựng các hàm thực hiện các công việc sau:

a. Nhập danh sách liên kết theo giải thuật thêm vào cuối danh sách, mỗi phần tử gồm có các thông tin sau: mssv (int), và hoten (char hoten[30]), dtb (float).

b. Liệt kê danh sách ra màn hình.

c. Cho biết tổng số sinh viên có trong danh sách, đặt tên hàm là Reccount.

d. Thêm 1 sinh viên vào sau phần tử có thứ tự thứ i trong danh sách.

Ghi chú:

- Thứ tự theo qui ước bắt đầu là 1
- Nếu (i = 0) thêm vào đầu danh sách
- Nếu i > Reccount(First) thì thêm vào cuối danh sách.

e. In ra họ tên của sinh viên có mã số được nhập vào.

f. Loại bỏ sinh viên có mã số được nhập vào (trước khi xóa hỏi lại "Bạn thật sự muốn xóa (Y/N)?")

g. Sắp xếp lại danh sách theo thứ tự đtb giảm dần.

11. Viết chương trình tạo một danh sách liên kết chứa tên học viên, điểm trung bình, hạng của học viên (với điều kiện chỉ nhập tên và điểm trung bình), quá trình nhập sẽ dừng lại khi tên nhập vào là rỗng. Xếp hạng cho các học viên. In ra danh sách học viên thứ tự hạng tăng dần (Ghi chú: Cùng điểm trung bình thì cùng hạng).

12. Viết chương trình nhập hai đa thức theo danh sách liên kết. In ra tích của hai đa thức này.

Ví dụ: Đa thức First1 : $2x^5 + 4x^2 - 1$

Đa thức First2 : $10x^7 - 3x^4 + x^2$

⇒ Kết quả in ra : $20x^{12} + 34x^9 - 8x^7 - 12x^6 + 7x^4 - x^2$

(***Ghi chú***: Không nhập và in ra các số hạng có hệ số bằng 0)

13. Viết hàm thêm phần tử có nội dung x vào danh sách liên kết có thứ tự tăng dần sao cho sau khi thêm danh sách liên kết vẫn có thứ tự tăng.

14. Sử dụng STACK viết các hàm sau:

a. Đổi số từ hệ thập phân (10) sang hệ nhị phân (2).

b. Định trị một biểu thức.

CHƯƠNG 4: CÂY

-----ଓ✧ଌ-----

1. TỔNG QUAN

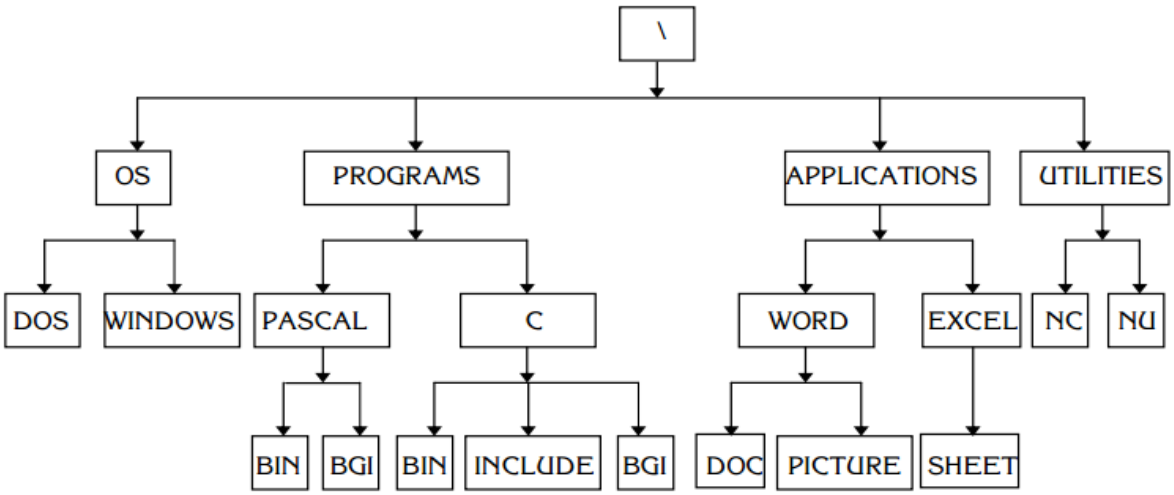
1.1. Định nghĩa

Cây là một tập hợp các phần tử (các nút) được tổ chức và có các đặc điểm sau:

- Hoặc là một tập hợp rỗng (cây rỗng)
- Hoặc là một tập hợp khác rỗng trong đó có một nút duy nhất được làm nút gốc (Root's Node), các nút còn lại được phân thành các nhóm trong đó mỗi nhóm lại là một cây gọi là cây con (Sub-Tree).

✧ Như vậy, một cây con có thể là một tập rỗng các nút và cũng có thể là một tập hợp khác rỗng trong đó có một nút làm nút gốc cây con.

Ví dụ: Cây thư mục trên một đĩa cứng



1.2 Một số khái niệm

1.2.1. Bậc của một nút

Bậc của một nút (node's degree) là số cây con của nút đó.

Ví dụ: Bậc của nút OS trong cây trên bằng 2.

1.2.2. Bậc của một cây

Bậc của một cây (tree's degree) là bậc lớn nhất của các nút trong cây.

Cây có bậc N gọi là cây N-phân (N-Tree)

Ví dụ: Bậc của cây trên bằng 4 (bằng bậc của nút gốc) và cây trên gọi là cây tứ phân (Quarter-Tree).

1.2.3. Nút gốc

Nút gốc (root's node) là nút không phải là nút gốc cây con của bất kỳ một cây con nào khác trong cây (nút không làm nút gốc cây con).

Ví dụ: Nút \ của cây trên là các nút gốc.

1.2.4. Nút kết thúc

Nút kết thúc hay còn gọi là nút lá (leaf's node) là nút có bậc bằng 0 (không có nút cây con).

Ví dụ: Các nút DOS, WINDOWS, BIN, INCLUDE, BGI, DOC, PICTURE, SHEET, NC, NU của cây trên là các nút lá.

1.2.5. Nút trung gian

Nút trung gian hay còn gọi là nút giữa (interior's node) là nút không phải là nút gốc và cũng không phải là nút kết thúc (nút có bậc khác không và là nút gốc cây con của một cây con nào đó trong cây).

Ví dụ: Các nút OS, PROGRAMS, APPLICATIONS, UTILITIES, PASCAL, C, WORD, EXCEL của cây trên là các nút trung gian.

1.2.6. Mức của một nút

Mức của một nút (node's level) bằng mức của nút gốc cây con chứa nó cộng thêm 1, trong đó mức của nút gốc bằng 1.

Ví dụ: Mức của các nút DOS, WINDOWS, PASCAL, C, WORD, EXCEL, NC, NU của cây trên bằng 3; mức của các nút BIN, INCLUDE, BGI, DOC, PICTURE, SHEET, của cây trên bằng 4.

1.2.7. Chiều cao hay chiều sâu của một cây

Chiều cao của một cây (tree's height) hay chiều sâu của một cây (tree's depth) là mức cao nhất của các nút trong cây.

Ví dụ: Chiều cao của cây trên bằng 4.

1.2.8. Nút trước và nút sau của một nút

Nút T được gọi là nút trước (ancestor's node) của nút S nếu cây con có gốc là T chứa cây con có gốc là S. Khi đó, nút S được gọi là nút sau (descendant's node) của nút T.

Ví dụ: Nút PROGRAMS là nút trước của các nút BIN, BGI, INCLUDE, PASCAL, C và ngược lại các nút BIN, BGI, INCLUDE, PASCAL, C là nút sau của nút PROGRAMS trong cây trên.

1.2.9. Nút cha và nút con của một nút

Nút B được gọi là nút cha (parent's node) của nút C nếu nút B là nút trước của nút C và mức của nút C lớn hơn mức của nút B là 1. Khi đó, nút C được gọi là nút con (child's node) của nút B.

Ví dụ: Nút PROGRAMS là nút cha của các nút PASCAL, C và ngược lại các nút PASCAL, C là nút con của nút PROGRAMS trong cây trên.

1.2.10. Nút anh em

Các nút có cùng một cha được gọi là các nút anh em.

Ví dụ: Nút WORD và nút EXCEL là 2 nút anh em. Hay nói cách khác WORD là nút anh em của nút EXCEL và EXCEL là nút anh em của nút WORD.

1.2.11. Chiều dài đường đi của một nút

Chiều dài đường đi của một nút là số đỉnh (số nút) tính từ nút gốc để đi đến nút đó. Chiều dài đường đi của nút gốc luôn luôn bằng 1, chiều dài đường đi tới một nút bằng chiều dài đường đi tới nút cha nó cộng thêm 1.

Ví dụ: Chiều dài đường đi tới nút PROGRAMS trong cây trên là 2.

1.2.12. Chiều dài đường đi của một cây

Chiều dài đường đi của một cây (path's length of the tree) là tổng tất cả các chiều dài đường đi của tất cả các nút trên cây.

Ví dụ: Chiều dài đường của cây trên là 65.

✧ ***Ghi chú***: Đây là chiều dài đường đi trong (internal path's length) của cây. Để có được chiều dài đường đi ngoài (external path's length) của cây người ta mở rộng tất cả các nút của cây sao cho tất cả các nút của cây có cùng bậc bằng cách thêm vào các nút giả sao cho tất cả các nút có bậc bằng bậc của cây. Chiều dài đường đi ngoài của cây bằng tổng chiều dài của tất cả các nút mở rộng.

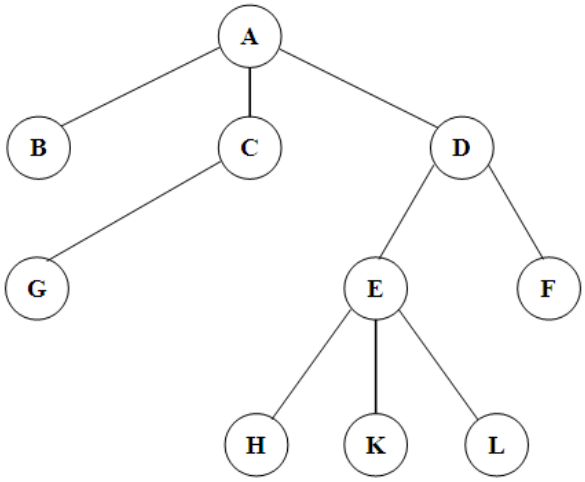
1.2.13. Rừng

Rừng (forest) là tập hợp các cây.

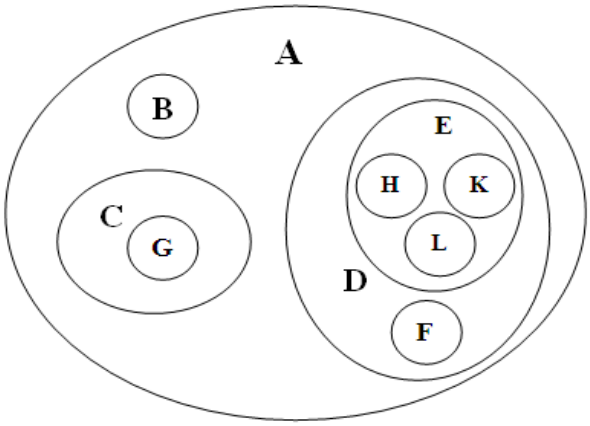
Như vậy, một cây khi mất nút gốc có thể sẽ trở thành một rừng.

1.3. Cách biểu diễn

1.3.1. Biểu diễn bằng đồ thị



1.3.2. Biểu diễn bằng giản đồ



1.3.3. Biểu diễn bằng phương pháp Indentation

A

- B

- C

- - G

- D

- - E

- - - H

- - - K

- - - L

- - F

1.3.4. Biểu diễn bằng chỉ số

1A

1.1B

1.2C

1.2.1G

1.3D

1.3.1E

1.3.1.1H

1.3.1.2K

1.3.1.3L

1.3.2F

1.3.5. Biểu diễn bằng cặp dấu ( )

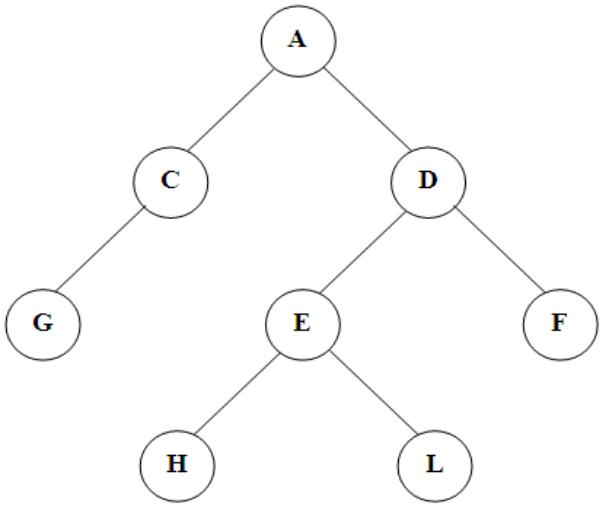
(A(B)(C(G))(D(E(H)(K)(L))(F)))

1.4. Lưu trữ trong bộ nhớ

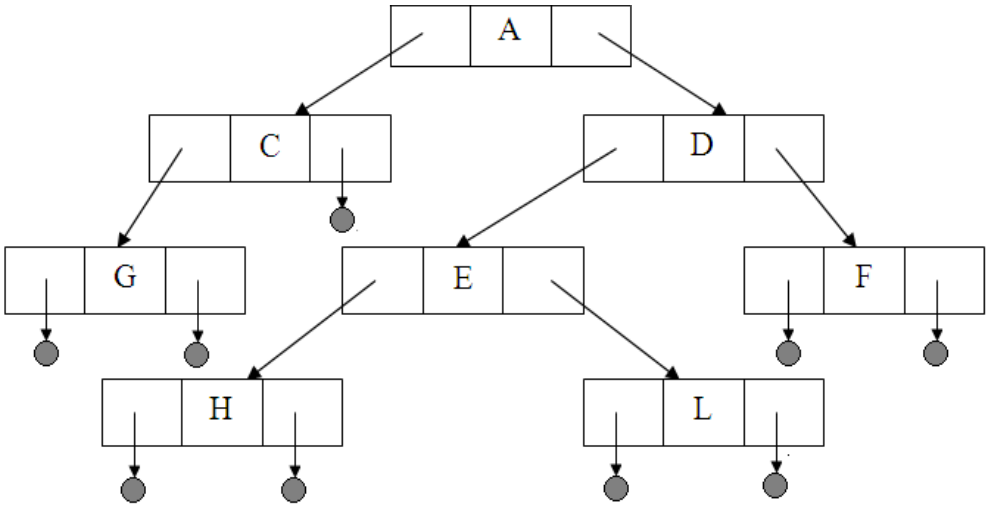
1.4.1. Cách tổng quát

Ta có thể sử dụng danh sách liên kết, trong trường hợp tổng quát ta dùng danh sách n_liên kết để chứa cây n_phân.

Ví dụ: Cây nhị phân



Được lưu trong bộ nhớ như sau:



Phương pháp này chỉ có lợi khi hầu hết các nút của cây có bậc là bậc của cây, tức là các vùng liên kết của các nút đều được sử dụng.

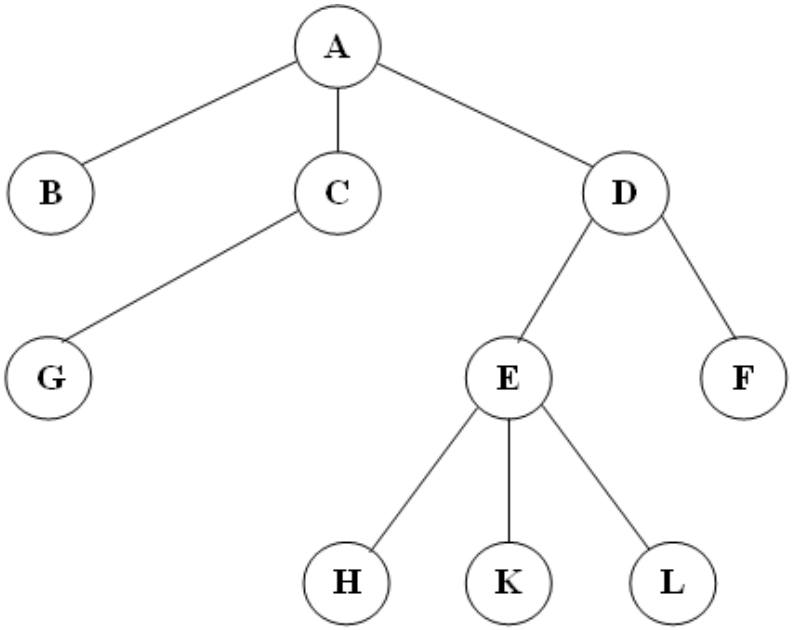
1.4.2. Thông qua cây nhị phân

Ta có thể mô tả và chứa cây n_phân bằng cách dùng cây nhị phân và chứa nó trong bộ nhớ theo qui ước:

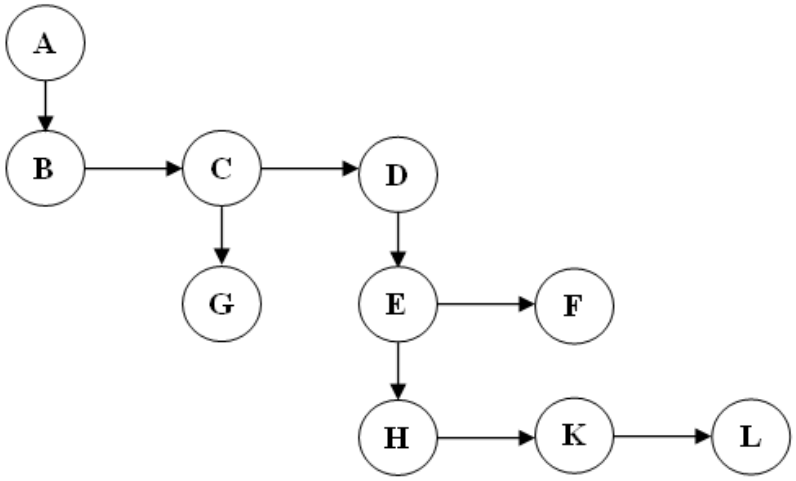
- Các nút có cùng cha (các nút anh em) sẽ ở trên cùng đường ngang.
- Các nút con ở đường phía dưới.

Khi đó đối với 1 nút thì liên kết trái chỉ đến nút con (mũi tên xuống) và liên kết phải chỉ đến 1 nút anh em (mũi tên ngang).

Ví dụ: Cây tam phân



Mô tả qua cây nhị phân như sau:



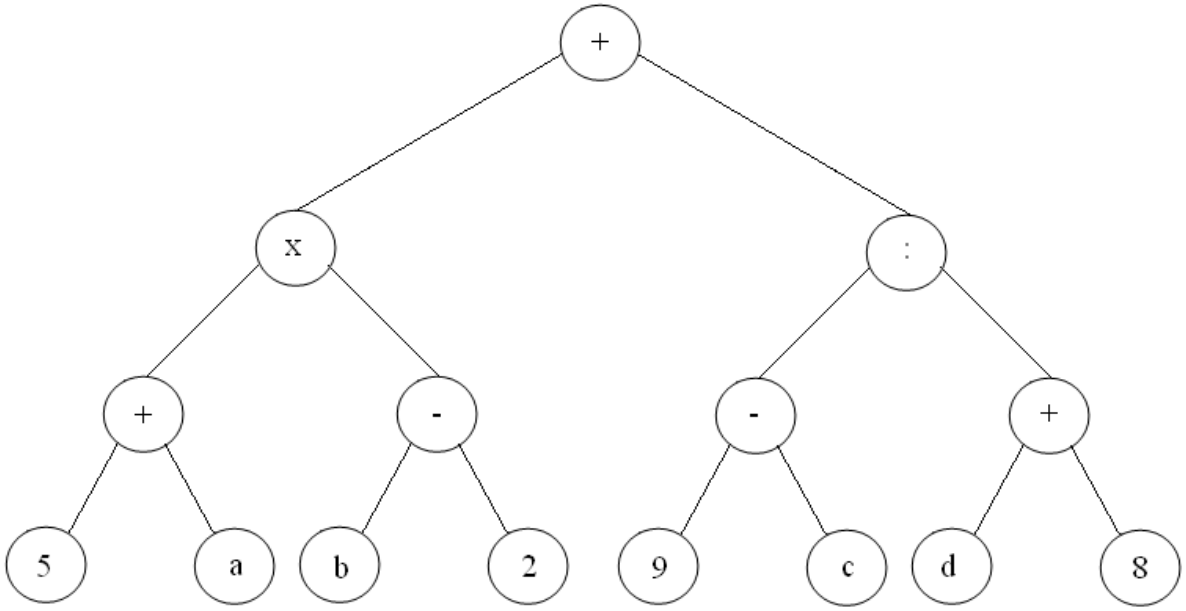
✧ Như vậy ta có thể biểu diễn cây n_phân thông qua cây nhị phân. Và trong phạm vi phần này chúng ta sẽ trình bày các thao tác trên ***cây nhị phân*** (Binary Tree) là cây phổ biến và thông dụng nhất.

2. CÂY NHỊ PHÂN

2.1. Định nghĩa

Cây nhị phân là cây có bậc bằng 2 (bậc của mỗi nút tối đa bằng 2). Hay nói cách khác, cây nhị phân là cây mà tại mỗi nút của nó có tối đa là 2 cây con.

Ví dụ: Cây nhị phân biểu diễn biểu thức [(5+a) x (b-2)] + [(9-c) : (d+8)] như sau:



2.2. Khai báo cấu trúc dữ liệu

Để biểu diễn cây nhị phân trong bộ nhớ máy tính chúng ta có thể sử dụng danh sách có 2 mối liên kết để quản lý địa chỉ của 2 nút gốc cây con (cây con trái và cây con phải).

Ta nhận thấy cấu trúc dữ liệu của cây nhị phân tương tự như cấu trúc dữ liệu của danh sách liên kết đôi nhưng về cách thức liên kết thì khác nhau:

```
typedef <Kiểu_dữ_liệu> Info;
struct BinT_Node
{
     Info Key;
     BinT_Node* Left; /* Vùng liên kết quản lý địa
                         chỉ nút gốc cây con trái*/
     BinT_Node* Right;/*Vùng liên kết quản lý địa
                         chỉ nút gốc cây con phải*/
};
typedef BinT_Node* BinT_Type;
```

2.3. Các phép toán

2.3.1. Khởi tạo

Việc khởi tạo cây nhị phân chỉ đơn giản là chúng ta cho con trỏ quản lý địa chỉ nút gốc về con trỏ NULL.

Hàm khởi tạo cây nhị phân như sau:

```
void Initialize(BinT_Type &BTree)
{
    BTree = NULL;
}
```

2.3.2. Tạo mới một nút

Thao tác này hoàn toàn tương tự như đối với thao tác tạo mới một nút trong danh sách liên kết đôi. Giả sử chúng ta cần tạo mới một nút có thành phần dữ liệu là NewData.

Thuật toán

B1: BTNode = new BinT_Node

B2: IF (BTNode = NULL)

Thực hiện Bkt

B3: BTNode -> Left = NULL

B4: BTNode -> Right = NULL

B5: BTNode -> Key = NewData

Bkt: Kết thúc

Cài đặt thuật toán

Hàm Create_Node có prototype:

```
BinT_Type Create_Node(Info  NewData);
```

Hàm tạo mới một nút có thành phần dữ liệu là NewData, hàm trả về con trỏ trỏ tới địa chỉ của nút mới tạo. Nếu không đủ bộ nhớ để tạo, hàm trả về con trỏ NULL.

```
BinT_Type Create_Node(Info NewData)
{
    BinT_Type BTnode = new BinT_Node;
    if(BTnode != NULL)
    {
         BTnode->Left = NULL;
         BTnode->Right = NULL;
         BTnode->Key = NewData;
     }
     return BTnode;
}
```

2.3.3. Thêm một nút

Giả sử chúng ta cần thêm một nút có giá trị thành phần dữ liệu là NewData vào trong cây nhị phân. Việc thêm có thể diễn ra ở cây con trái hoặc cây con phải của cây nhị phân. Do vậy, ở đây chúng ta trình bày 2 thao tác thêm riêng biệt nhau:

Thuật toán thêm 1 nút vào bên trái nhất của cây

B1: NewNode = Create_Node (NewData)

B2: IF (NewNode = NULL)

Thực hiện Bkt

B3: IF (BTree = NULL) *// Cây rỗng*

B3.1: BTree = NewNode

B3.2: Thực hiện Bkt

B4: Lnode = BTree

B5: IF (Lnode -> Left = NULL) *// Cây con trái rỗng*

B5.1: Lnode -> Left = NewNode

B5.2: Thực hiện Bkt

B6: Lnode = Lnode -> Left *// Đi theo nhánh cây con trái*

B7: Lặp lại B5

Bkt: Kết thúc

Cài đặt thuật toán

Hàm Add_Left có prototype:

```
void Add_Left(BinT_Type &BTree, Info NewData);
```

Hàm thực hiện việc thêm vào bên trái nhất trong cây nhị phân BTree một nút có thành phần dữ liệu là NewData, hàm trả về con trỏ trỏ tới địa chỉ của nút mới thêm nếu việc thêm thành công, ngược lại nếu không đủ bộ nhớ, hàm trả về con trỏ NULL.

```
void Add_Left(BinT_Type &BTree, Info NewData)
{
     BinT_Type NewNode = Create_Node(NewData);
     if(BTree == NULL)
          BTree = NewNode;
     else{
          BinT_Type Lnode = BTree;
          while(Lnode->Left != NULL)
               Lnode = Lnode->Left;
          Lnode->Left = NewNode;
     }
}
```

Thuật toán thêm 1 nút vào bên phải nhất của cây

Hoàn toàn tương tự thuật toán thêm 1 nút vào bên phải nhất của cây.

2.3.4. Duyệt qua các nút trên cây

Trong thao tác này chúng ta tìm cách duyệt qua (ghé thăm) tất cả các nút trong cây nhị phân để thực hiện một thao tác xử lý nào đó đối với nút này (xem nội dung thành phần dữ liệu chẳng hạn). Căn cứ vào thứ tự duyệt nút gốc so với 2 nút gốc cây con, thao tác duyệt có thể thực hiện theo các cách khác nhau.

Các cách duyệt cây

- Duyệt theo thứ tự nút gốc trước (Preorder – Tiền tự)

Theo cách duyệt này thì nút gốc sẽ được duyệt trước sau đó mới duyệt đến hai cây con. Dựa vào thứ tự duyệt hai cây con mà chúng ta có hai cách duyệt theo thứ tự nút gốc trước:

+ Duyệt nút gốc, duyệt cây con trái, duyệt cây con phải (Root – Left – Right)

+ Duyệt nút gốc, duyệt cây con phải, duyệt cây con trái (Root – Right – Left)

- Duyệt theo thứ tự nút gốc giữa (Inorder – Trung tự)

Theo cách duyệt này thì chúng ta duyệt một trong hai cây con trước rồi đến duyệt nút gốc và sau đó mới duyệt cây con còn lại. Căn cứ vào thứ tự duyệt hai cây con chúng ta cũng sẽ có hai cách duyệt theo thứ tự nút gốc giữa:

+ Duyệt cây con trái, duyệt nút gốc, duyệt cây con phải (Left – Root – Right)

+ Duyệt cây con phải, duyệt nút gốc, duyệt cây con trái (Right – Root – Left)

- Duyệt theo thứ tự nút gốc sau (Postorder – Hậu tự)

Tương tự như duyệt theo nút gốc trước, trong cách duyệt này thì nút gốc sẽ được duyệt sau cùng so với duyệt hai nút gốc cây con.

Do vậy, căn cứ vào thứ tự duyệt hai cây con mà chúng ta cũng có hai cách duyệt theo thứ tự nút gốc sau:

+ Duyệt cây con trái, duyệt cây con phải, duyệt nút gốc (Left – Right – Root)

+ Duyệt cây con phải, duyệt cây con trái, duyệt nút gốc (Right – Left – Root)

Trong phần này chúng ta chỉ trình bày một cách duyệt theo một thứ tự cụ thể đó là: Duyệt cây con trái, duyệt nút gốc và duyệt cây con phải (Left – Root – Right) và sử dụng thuật toán đệ qui. Các cách duyệt khác bằng thuật toán đệ qui hay không đệ qui sinh viên tự vận dụng tương tự.

Thuật toán đệ qui để duyệt cây nhị phân theo thứ tự Left – Root – Right

B1: CurNode = BTree

B2: IF (CurNode = NULL)

Thực hiện Bkt

B3: LRootR (BTree -> Left) *// Duyệt cây con trái*

B4: Process (CurNode -> Key) *// Xử lý thông tin nút gốc*

B5: LRootR (BTree -> Right) *// Duyệt cây con phải*

Bkt: Kết thúc

Cài đặt thuật toán

Hàm LRootR_Traverse có prototype:

```
void LRootR_Traverse(BinT_Type BTree);
```

Hàm thực hiện thao tác duyệt qua tất cả các nút trong cây nhị phân BTree theo thứ tự duyệt Left – Root – Right để xử lý thông tin ở mỗi nút là in ra khóa của nút.

```
void LRootR_Traverse(BinT_Type BTree)
{    if(BTree == NULL)
          return;
     LRootR_Traverse(BTree->Left);
     Process(BTree->Key);
     LRootR_Traverse(BTree->Right);
}
```

✧ ***Lưu ý***: Hàm Process thực hiện việc xử lý thông tin (Key) của mỗi nút. Do vậy tùy từng trường hợp cụ thể mà chúng ta viết hàm cho phù hợp. Chẳng hạn để xuất thông tin thì chỉ cần các lệnh xuất dữ liệu để xuất thành phần Key là `cout << BTree->Key << '\t';`.

2.3.5. Tính chiều cao của cây

Để tính chiều cao của cây (TH) chúng ta phải tính chiều cao của các cây con, khi đó chiều cao của cây chính là chiều cao lớn nhất của các cây con cộng thêm 1 (chiều cao nút gốc). Như vậy thao tác tính chiều cao của cây là thao tác tính đệ qui chiều cao của các cây con (chiều cao của cây con có gốc là nút lá bằng 1).

Thuật toán

B1: IF (BTree = NULL)

B1.1: TH = 0

B1.2: Thực hiện Bkt

B2: THL = TH(BTree -> Left)

B3: THR = TH(BTree -> Right)

B4: IF (THL > THR)

TH = THL + 1

B5: ELSE

TH = THR + 1

Bkt: Kết thúc

Cài đặt thuật toán

Hàm Tree_Height có prototype:

```
int Tree_Height(BinT_Type BTree);
```

Hàm tính chiều cao của cây BTree theo thuật toán đệ qui. Hàm trả về chiều cao của cây cần tính.

```
int Tree_Height(BinT_Type BTree)
{
     if(BTree == NULL)
     return 0;
     int HTL = Tree_Height(BTree->Left);
     int HTR = Tree_Height(BTree->Right);
     if(HTL > HTR)
          return HTL + 1;
     else
          return HTR + 1;
}
```

2.3.6. Đếm số nút của cây

Tương tự như tính chiều cao của cây, số nút của cây (NN) bằng tổng số nút của hai cây con cộng thêm 1. Do vậy thao tác này chúng ta cũng sẽ tính đệ qui số nút của các cây con (số nút của cây con có gốc là nút lá bằng 1).

Thuật toán

B1: IF (BTree = NULL)

B1.1: NN = 0

B1.2: Thực hiện Bkt

B2: NNL = NN(BTree -> Left)

B3: NNR = NN(BTree -> Right)

B4: NN = NNL + NNR + 1

Bkt: Kết thúc

Cài đặt thuật toán

Hàm Count_Node có prototype:

```
int Count_Node(BinT_Type BTree);
```

Hàm đếm số nút của cây BTree theo thuật toán đệ qui.

Hàm trả về tổng số nút của cây.

```
int Count_Node(BinT_Type BTree)
{
     if(BTree == NULL) return 0;
     int NNL = Count_Node(BTree->Left);
     int NNR = Count_Node(BTree->Right);
     return(NNL + NNR + 1);
}
```

2.3.7. Xóa một nút

Việc xóa một nút trong cây có thể làm cho cây trở thành rừng. Do vậy trong thao tác này nếu chúng ta tiến hành xóa một nút lá thì không có điều gì xảy ra, song nếu xóa nút không phải là nút lá thì chúng ta phải tìm cách chuyển các nút gốc cây con là các nút con của nút cần xóa thành các nút gốc cây con của các nút khác rồi mới tiến hành hủy nút này.

- Trường hợp nếu nút cần xóa chỉ có 1 nút gốc cây con thì chúng ta có thể chuyển nút gốc cây con này thành nút gốc cây con của nút cha của nút cần xóa.
- Trường hợp nếu nút cần xóa có 2 nút gốc cây con thì chúng ta phải chuyển 2 nút gốc cây con này thành nút gốc cây con của các nút khác với nút cần xóa. Việc chọn các nút để làm nhiệm vụ nút cha của các nút gốc cây con này tùy vào từng trường hợp cụ thể của cây nhị phân mà chúng ta sẽ lựa chọn cho phù hợp.

Do vậy, thao tác xóa một nút sẽ được trình bày cụ thể trong các loại cây cụ thể được trình bày ở các phần sau chẳng hạn như cây nhị phân tìm kiếm.

Để có thể thực thi được các hàm trên ta cần xây dựng thêm hàm nhập cây nhị phân và hàm main() để gọi thực hiện chúng.

2.3.8. Nhập cây nhị phân

```
void Read_Tree(BinT_Type &BTree)
{    Info X;
     cin>>X;
     if(X!=0){
          BinT_Type BT;
          BT = Create_Node(X);
          BTree = BT;
          cout<<"Nhap con trai cua "<<X<<" : ";
          Read_Tree(BTree->Left);
          cout<<"Nhap con phai cua "<<X<<" : ";
          Read_Tree(BTree->Right);
     }
     else BTree=NULL;
}
```

2.3.9. Hàm main()

```
main()
{    BinT_Type BT;
     cout<<"=====NHAP CAY NHI PHAN====="<<endl;
     Initialize(BT);
     Read_Tree(BT);
     cout<<"=====IN CAY NHI PHAN THEO CACH LNR====="<<endl;
     LRootR_Traverse(BT);
     cout<<endl;
     Info X;
     cout<<"Ban muon them vao nut trai nhat la bao nhieu:";
     cin>>X;
     Add_Left(BT, X);
     cout<<"==CAY NHI PHAN SAU KHI THEM NUT TRAI=="<<endl;
     LRootR_Traverse(BT);
     cout<<"\nChieu cao cua cay la: "<<Tree_Height(BT);
     cout<<"\nCay co "<<Count_Node(BT)<<" nut"<<endl;
}
```

3. CÂY NHỊ PHÂN TÌM KIẾM

3.1. Định nghĩa

Cây nhị phân tìm kiếm (BST – Binary Search Tree) là cây nhị phân mà tại mỗi nút của cây thì khóa của nút này lớn hơn khóa của tất cả các nút thuộc cây con bên trái và nhỏ hơn khóa của tất cả các nút thuộc cây con bên phải.

Gọi: K_N là khóa của nút gốc.

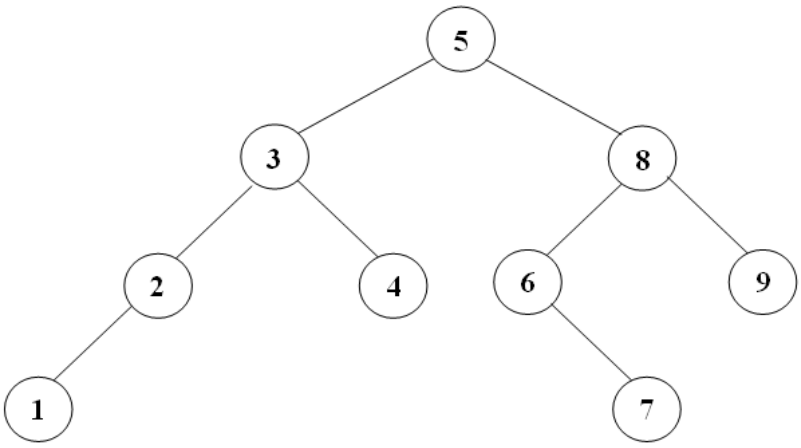
K_L là khóa của nút con trái.

K_R là khóa của nút con phải.

Ta có: $K_L < K_N < K_R$

Như vậy theo định nghĩa trên thì trong cây nhị phân tìm kiếm không có các nút trùng khóa với nhau và khi duyệt cây theo cách LNR ta sẽ được 1 dãy có thứ tự tăng dần.

Ví dụ:



3.2. Cấu trúc dữ liệu

- Cấu trúc dữ liệu của cây nhị phân tìm kiếm là cấu trúc dữ liệu để biểu diễn các cây nhị phân nói chung.

```
typedef <Kiểu_dữ_liệu> Info;
struct BST_Node
{    Info Key;
     BST_Node* Left;
     BST_Node* Right;
};
typedef BST_Node* BST_Type;
```

- Khóa nhận diện (Key) của các nút trong cây nhị phân tìm kiếm đôi một khác nhau (không có hiện tượng trùng khóa). Tuy nhiên trong trường hợp cần quản lý các nút có khóa trùng nhau trong cây nhị phân tìm kiếm thì chúng ta có thể mở rộng cấu trúc dữ liệu của mỗi nút bằng cách thêm thành phần Count để ghi nhận số lượng các nút trùng khóa. Khi đó, cấu trúc dữ liệu để quản lý các cây nhị phân tìm kiếm được mở rộng như sau:

```
struct BST_Node
{
     Info Key;
     int Count;
     BST_Node* Left;
     BST_Node* Right;
  };
  typedef BST_Node* BST_Type;
```

- Nút ở bên trái nhất là nút có giá trị khóa nhận diện nhỏ nhất và nút ở bên phải nhất là nút có giá trị khóa nhận diện lớn nhất trong cây nhị phân tìm kiếm.
- Trong một cây nhị phân tìm kiếm thứ tự duyệt cây Left - Root - Right là thứ tự duyệt theo sự tăng dần các giá trị của Key trong các nút và thứ tự duyệt cây Right - Root - Left là thứ tự duyệt theo sự giảm dần các giá trị của Key trong các nút.

3.3. Các phép toán

3.3.1. Tìm kiếm một nút

Giả sử chúng ta cần tìm trên cây nhị phân tìm kiếm xem có tồn tại nút có khóa Key là SearchData hay không.

Để thực hiện thao tác này chúng ta sẽ vận dụng thuật toán tìm kiếm nhị phân: Do đặc điểm của cây nhị phân tìm kiếm thì tại một nút, nếu Key của nút này khác với SearchData thì SearchData chỉ có thể tìm thấy hoặc trên cây con trái của nút này nếu SearchData nhỏ hơn Key của nút này hoặc trên cây con phải của nút này nếu SearchData lớn hơn Key của nút này.

Thuật toán

B1: CurNode = BST

B2: IF (CurNode = NULL) or (CurNode->Key = SearchData)

Thực hiện Bkt

B3: IF (CurNode -> Key > SearchData) *// Tìm kiếm trên cây con trái*

CurNode = CurNode -> Left

B4: ELSE *// Tìm kiếm trên cây con phải*

CurNode = CurNode -> Right

B5: Lặp lại B2

Bkt: Kết thúc

Cài đặt thuật toán

Hàm BST_Searching có prototype:

```
BST_Type Searching(BST_Type BST,Info SearchData);
```

Hàm thực hiện thao tác tìm kiếm trên cây nhị phân tìm kiếm BST nút có thành phần Key là SearchData. Hàm trả về con trỏ trỏ tới địa chỉ của nút có Key là SearchData nếu tìm thấy, trong trường hợp ngược lại hàm trả về con trỏ NULL.

```
BST_Type Searching(BST_Type BST, Info SearchData)
{    BST_Type CurNode = BST;
     while(CurNode != NULL && CurNode->Key != SearchData)
          if(CurNode->Key > SearchData)
               CurNode = CurNode->Left;
          else
               CurNode = CurNode->Right;
     return CurNode;
}
```

3.3.2. Thêm một nút

Giả sử chúng ta cần thêm một nút có thành phần dữ liệu (Key) là NewData vào trong cây nhị phân tìm kiếm sao cho sau khi thêm cây vẫn là một cây nhị phân tìm kiếm. Trong thao tác này trước hết chúng ta phải tìm kiếm vị trí thêm, sau đó mới tiến hành thêm nút mới vào cây. Quá trình tìm kiếm tuân thủ các bước trong thuật toán tìm kiếm đã trình bày ở trên.

Trong thuật toán này chúng ta sẽ trình bày thao tác thêm vào cây nhị phân tìm kiếm trong trường hợp không có hiện tượng trùng khóa. Do vậy, nếu NewData bị trùng với Key của một trong các nút ở trong cây nhị phân tìm kiếm thì chúng ta sẽ không thực hiện thao tác thêm này. Tuy nhiên, nếu chúng ta sử dụng cấu trúc dữ liệu mở rộng thì việc trùng khóa sẽ giải quyết đơn giản vì không làm tăng số nút của cây nhị phân tìm kiếm mà chỉ làm tăng thành phần Count của nút bị trùng khóa thêm 1.

Thuật toán

B1: IF (BST = NULL) *// Cây rỗng*

B1.1: BST = **new** BST_Node

B1.2: IF(BST != NULL)

B1.2.1: BST->Left = NULL;

B1.2.2: BST->Right = NULL;

B1.2.3: BST->Key = NewData;

B1.3: Thực hiện Bkt

B2: ELSE *//BST != NULL*

B2.1: IF (BST->Key = NewData) *// Trùng khóa*

Thực hiện Bkt

B2.2: IF (BST -> Key > NewData)

Đệ qui trái *// Thêm NewData vào cây con trái của BST*

B2.3: ELSE (BST -> Key < NewData)

Đệ qui phải *// Thêm NewData vào cây con phải của BST*

Bkt: Kết thúc

Cài đặt thuật toán

Hàm BST_Add_Node có prototype:

```
BST_Type  BST_Add_Node(BST_Type  &BSTree,Info  NewData);
```

Hàm thực hiện việc thêm vào cây nhị phân tìm kiếm BSTree một nút có thành phần Key là NewData. Hàm trả về con trỏ trỏ tới địa chỉ của nút mới thêm nếu việc thêm thành công, trong trường hợp ngược lại hàm trả về con trỏ NULL.

```
void Add_Node(BST_Type  &BST, Info  NewData)
{
     if(BST == NULL){
          BST = new BST_Node;
          if(BST != NULL){
               BST->Left = NULL;
               BST->Right = NULL;
               BST->Key = NewData;
          }
     }
     else  //BST != NULL
          if(BST->Key == NewData) return;
          else
               if(BST->Key > NewData)
                    Add_Node(BST->Left, NewData);
               else
                    Add_Node(BST->Right, NewData);
}
```

3.3.3. Xóa một nút

Cũng như thao tác thêm một nút vào trong cây nhị phân tìm kiếm, thao tác xóa một nút trên cây nhị phân tìm kiếm cũng phải bảo đảm cho cây sau khi xóa nút đó thì cây vẫn là một cây nhị phân tìm kiếm. Đây là một thao tác không đơn giản bởi nếu không cẩn thận chúng ta sẽ biến cây thành một rừng.

Giả sử chúng ta cần xóa nút có thành phần dữ liệu (Key) là DelData ra khỏi cây nhị phân tìm kiếm. Điều đầu tiên trong thao tác này là chúng ta phải tìm kiếm địa chỉ của nút cần xóa là DelNode, sau đó mới tiến hành xóa nút có địa chỉ là DelNode này nếu tìm thấy (do đó thuật toán này còn được gọi là thuật toán tìm kiếm và loại bỏ trên cây). Quá trình tìm kiếm đã trình bày ở trên, ở đây chúng ta chỉ trình bày thao tác xóa khi tìm thấy nút có địa chỉ DelNode (DelNode -> Key = DelData) và trong quá trình tìm kiếm chúng ta giữ địa chỉ nút cha của nút cần xóa là PrDelNode.

Việc xóa nút có địa chỉ DelNode có thể xảy ra một trong ba trường hợp sau: DelNode là nút lá, DelNode là nút chỉ có 1 nút gốc cây con, DelNode là nút có đủ 2 nút gốc cây con.

DelNode là nút lá

Trong trường hợp này đơn giản là chúng ta chỉ cần cắt bỏ mối quan hệ cha-con giữa PrDelNode và DelNode bằng cách cho con trỏ PrDelNode -> Left (nếu DelNode là nút con bên trái của PrDelNode) hoặc cho con trỏ PrDelNode -> Right (nếu DelNode là nút con bên phải của PrDelNode) về con trỏ NULL và tiến hành xóa (delete) nút có địa chỉ DelNode này.

DelNode là nút chỉ có 1 nút gốc cây con

Trong trường hợp này cũng khá đơn giản, chúng ta chỉ cần chuyển mối quan hệ cha-con giữa PrDelNode và DelNode thành mối quan hệ cha-con giữa PrDelNode và nút gốc cây con của DelNode rồi tiến hành cắt bỏ mối quan hệ cha-con giữa DelNode và 1 nút gốc cây con của nó và tiến hành xóa nút có địa chỉ DelNode này.

DelNode là nút có đủ 2 nút gốc cây con

Trường hợp này khá phức tạp, việc xóa có thể tiến hành theo một trong hai cách sau đây (có thể có nhiều cách khác nữa song ở đây chúng ta chỉ nghiên cứu hai cách):

✧ ***Cách 1***: Chuyển 2 cây con của DelNode về thành một cây con

Theo phương pháp này chúng ta sẽ chuyển cây con phải của DelNode (DelNode -> Right) về thành cây con phải của cây con có nút gốc là nút phải nhất trong cây con trái của DelNode (phải nhất trong DelNode -> Left), hoặc chuyển cây con trái của DelNode (DelNode -> Left) về thành cây con trái của cây con có nút gốc là nút trái nhất trong cây con phải của DelNode (trái nhất trong DelNode -> Right). Sau khi chuyển thì DelNode sẽ trở thành nút chỉ có 1 cây con và chúng ta xóa DelNode như đối với trường hợp *DelNode là nút chỉ có 1 nút gốc cây con*.

✧ ***Cách 2***: Sử dụng phần tử thế mạng (standby)

Theo phương pháp này chúng ta sẽ không xóa nút có địa chỉ DelNode mà chúng ta sẽ xóa nút có địa chỉ của phần tử thế mạng là nút phải nhất trong cây con trái của DelNode (MRNode), hoặc là nút trái nhất trong cây con phải của DelNode (MLNode). Sau khi chuyển toàn bộ nội dung dữ liệu của nút thế mạng cho DelNode (DelNode -> Key = MRNode -> Key hoặc DelNode -> Key = MLNode -> Key) thì chúng ta sẽ xóa nút thế mạng như đối với trường hợp *DelNode là nút lá* hoặc *DelNode là nút chỉ có 1 nút gốc cây con*.

Thuật toán xóa một nút trong cây BST bằng phương pháp chuyển cây

Sử dụng phương pháp chuyển cây con phải của nút cần xóa về thành cây con phải của cây con có nút gốc là nút phải nhất trong cây con trái của nút cần xóa (nếu nút cần xóa có đủ 2 cây con):

// Tìm nút cần xóa và nút cha của nút cần xóa

B1: DelNode = BSTree

B2: PrDelNode = NULL

B3: IF (DelNode = NULL)

Thực hiện Bkt

B4: IF (DelNode -> Key = DelData)

Thực hiện B8

B5: IF (DelNode -> Key > DelData) *// Chuyển sang cây con trái*

B5.1: PrDelNode = DelNode

B5.2: DelNode = DelNode -> Left

B5.3: OnTheLeft = True

B5.4: Thực hiện B7

B6: IF (DelNode -> Key < DelData) *// Chuyển sang cây con phải*

B6.1: PrDelNode = DelNode

B6.2: DelNode = DelNode -> Right

B6.3: OnTheLeft = False

B6.4: Thực hiện B7

B7: Lặp lại B3

// Chuyển các mối quan hệ của DelNode cho các nút khác

B8: IF (PrDelNode = NULL) // DelNode là nút gốc

// Nếu DelNode là nút lá

B8.1: If (DelNode -> Left = NULL) and (DelNode -> Right = NULL)

B8.1.1: BSTree = NULL

B8.1.2: Thực hiện B10

// Nếu DelNode có một cây con phải

B8.2: If (DelNode -> Left = NULL) and (DelNode -> Right != NULL)

B8.2.1: BSTree = BSTree -> Right

B8.2.2: DelNode -> Right = NULL

B8.2.3: Thực hiện B10

// Nếu DelNode có một cây con trái

B8.3: If (DelNode -> Left != NULL) and (DelNode -> Right = NULL)

B8.3.1: BSTree = BSTree -> Left

B8.3.2: DelNode -> Left = NULL

B8.3.3: Thực hiện B10

// Nếu DelNode có hai cây con

B8.4: If (DelNode -> Left != NULL) and (DelNode -> Right != NULL)

// Tìm nút phải nhất trong cây con trái của DelNode

B8.4.1: MRNode = DelNode -> Left

B8.4.2: if (MRNode -> Right = NULL)

Thực hiện B8.4.5

B8.4.3: MRNode = MRNode -> Right

B8.4.4: Lặp lại B8.4.2

// Chuyển cây con phải của DelNode về cây con phải của MRNode

B8.4.5: MRNode -> Right = DelNode -> Right

B8.4.6: DelNode -> Right = NULL

// Chuyển cây con trái còn lại của DelNode về cho BSTree

B8.4.7: BSTree = BSTree -> Left

B8.4.8: DelNode -> Left = NULL

B8.4.9: Thực hiện B10

B9: ELSE *// DelNode không phải là nút gốc*

// Nếu DelNode là nút lá

B9.1: If (DelNode -> Left = NULL) and (DelNode -> Right = NULL)

// DelNode là cây con trái của PrDelNode

B9.1.1: if (OnTheLeft = True)

PrDelNode -> Left = NULL

B9.1.2: else *// DelNode là cây con phải của PrDelNode*

PrDelNode -> Right = NULL

B9.1.3: Thực hiện B10

// Nếu DelNode có một cây con phải

B9.2: If (DelNode -> Left = NULL) and (DelNode -> Right != NULL)

B9.2.1: if (OnTheLeft = True)

PrDelNode -> Left = DelNode -> Right

B9.2.2: else

PrDelNode -> Right = DelNode -> Right

B9.2.3: DelNode -> Right = NULL

B9.2.4: Thực hiện B10

// Nếu DelNode có một cây con trái

B9.3: If (DelNode -> Left != NULL) and (DelNode -> Right = NULL)

B9.3.1: if (OnTheLeft = True)

PrDelNode -> Left = DelNode -> Left

B9.3.2: else

PrDelNode -> Right = DelNode -> Left

B9.3.3: DelNode -> Left = NULL

B9.3.4: Thực hiện B10

// Nếu DelNode có hai cây con

B9.4: If (DelNode -> Left != NULL) and (DelNode -> Right != NULL)

// Tìm nút phải nhất trong cây con trái của DelNode

B9.4.1: MRNode = DelNode -> Left

B9.4.2: if (MRNode -> Right = NULL)

Thực hiện B9.4.5

B9.4.3: MRNode = MRNode -> Right

B9.4.4: Lặp lại B9.4.2

// Chuyển cây con phải DelNode về thành cây con phải MRNode

B9.4.5: MRNode -> Right = DelNode -> Right

B9.4.6: DelNode -> Right = NULL

// Chuyển cây con trái còn lại của DelNode về cho PrDelNode

B9.4.7: if (OnTheLeft = True)

PrDelNode -> Left = DelNode -> Left

B9.4.8: else

PrDelNode -> Right = DelNode -> Left

B9.4.9: DelNode -> Left = NULL

B9.4.10: Thực hiện B10

// Hủy DelNode

B10: delete DelNode

Bkt: Kết thúc

Cài đặt thuật toán xóa một nút trong cây BST bằng phương pháp chuyển cây

Hàm Delete_Node_TRF có prototype:

```
int Delete_Node_TRF(BST_Type &BST, Info DelData);
```

Hàm thực hiện việc xóa nút có thành phần Key là DelData trên cây nhị phân tìm kiếm BSTree bằng phương pháp chuyển cây con phải của nút cần xóa về thành cây con phải của cây có nút gốc là nút phải nhất trong cây con trái của nút cần xóa (nếu nút cần xóa có hai cây con). Hàm trả về giá trị 1 nếu việc xóa thành công (có nút để xóa), trong trường hợp ngược lại hàm trả về giá trị 0 (không tồn tại nút có Key là DelData hoặc cây rỗng).

```
int Delete_Node_TRF(BST_Type &BST, Info DelData)
{
    BST_Type DelNode = BST;
    BST_Type PrDelNode = NULL;
    int OnTheLeft = 0;
    while(DelNode != NULL){
        if(DelNode->Key == DelData)
            break;
        PrDelNode = DelNode;
        if(DelNode->Key > DelData){
            DelNode = DelNode->Left;
            OnTheLeft = 1;
        }
        else // (DelNode->Key < DelData)
        {
            DelNode = DelNode->Right;
            OnTheLeft = 0;
        }
    }
    if(DelNode == NULL) // Khong co nut de huy
        return 0;
```

```
    if(PrDelNode == NULL)     // DelNode la nut goc
    {
      if(DelNode->Left==NULL && DelNode->Right==NULL)
            BST = NULL;
      else
            if(DelNode->Left == NULL)
                              // DelNode co 1 cay con phai
            {
                BST = BSTree->Right;
                DelNode->Right = NULL;
            }
            else
                if(DelNode->Right == NULL)
                            // DelNode co 1 cay con trai
                {
                    BST = BST->Left;
                    DelNode->Left = NULL;
                }
                else // DelNode co hai cay con
                {
                    BST_Type  MRNode = DelNode->Left;
                    while(MRNode->Right != NULL)
                        MRNode = MRNode->Right;
                    MRNode->Right = DelNode->Right;
                    DelNode->Right = NULL;
                    BST = BST->Left;
                    DelNode->Left = NULL;
                }
    }
    else    // DelNode khong la nut goc
    {
      if(DelNode->Left==NULL && DelNode->Right==NULL) //lá
```

```
            if(OnTheLeft == 1)
                PrDelNode->Left = NULL;
            else
                PrDelNode->Right = NULL;
        else
            if(DelNode->Left == NULL)
                                // DelNode co 1 cay con phai
            {
                if(OnTheLeft == 1)
                 PrDelNode->Left = DelNode->Right;
                else
                 PrDelNode->Right = DelNode->Right;
                DelNode->Right = NULL;
            }
            else
                if(DelNode->Right == NULL)
                                // DelNode co 1 cay con trai
                {
                    if(OnTheLeft == 1)
                      PrDelNode->Left = DelNode->Left;
                    else
                      PrDelNode->Right = DelNode->Left;
                    DelNode->Left = NULL;
                }
                else // DelNode co hai cay con
                {
                    BST_Type  MRNode = DelNode->Left;
                    while(MRNode->Right != NULL)
                        MRNode = MRNode->Right;
                    MRNode->Right = DelNode->Right;
                    DelNode->Right = NULL;
                    if(OnTheLeft == 1)
                      PrDelNode->Left = DelNode->Left;
```

```
                    else
                      PrDelNode->Right = DelNode->Left;
                    DelNode->Left = NULL;
                }
    }
    delete DelNode;
    return 1;
}
```

Thuật toán xóa một nút trong cây BST bằng phương pháp thế mạng

Sử dụng phương pháp xóa phần tử thế mạng là phần tử trái nhất (nhỏ nhất) trong cây con phải của nút cần xóa (nếu nút cần xóa có đủ 2 cây con).

// Tìm nút cần xóa và nút cha của nút cần xóa

B1: DelNode = BST

B2: PrDelNode = NULL

B3: IF (DelNode = NULL)

Thực hiện Bkt

B4: IF (DelNode->Key = DelData)

Thực hiện B8

B5: IF (DelNode -> Key > DelData) *// Chuyển sang cây con trái*

B5.1: PrDelNode = DelNode

B5.2: DelNode = DelNode -> Left

B5.3: OnTheLeft = True

B5.4: Thực hiện B7

B6: IF (DelNode -> Key < DelData) *// Chuyển sang cây con phải*

B6.1: PrDelNode = DelNode

B6.2: DelNode = DelNode -> Right

B6.3: OnTheLeft = False

B6.4: Thực hiện B7

B7: Lặp lại B3

// Chuyển các mối quan hệ của DelNode cho các nút khác

B8: IF (PrDelNode = NULL) // DelNode là nút gốc

// Nếu DelNode là nút lá

B8.1: If (DelNode -> Left = NULL) and (DelNode -> Right = NULL)

B8.1.1: BST = NULL

B8.1.2: Thực hiện B11

// Nếu DelNode có một cây con phải

B8.2: If (DelNode -> Left = NULL) and (DelNode -> Right != NULL)

B8.2.1: BST = BST -> Right

B8.2.2: DelNode -> Right = NULL

B8.2.3: Thực hiện B11

// Nếu DelNode có một cây con trái

B8.3: If (DelNode -> Left != NULL) and (DelNode -> Right = NULL)

B8.3.1: BST = BST -> Left

B8.3.2: DelNode -> Left = NULL

B8.3.3: Thực hiện B11

B9: ELSE *// DelNode không phải là nút gốc*

// Nếu DelNode là nút lá

B9.1: If (DelNode -> Left = NULL) and (DelNode -> Right = NULL)

// DelNode là cây con trái của PrDelNode

B9.1.1: if (OnTheLeft = True) PrDelNode -> Left = NULL

B9.1.2: else *// DelNode là cây con phải của PrDelNode*

PrDelNode -> Right = NULL

B9.1.3: Thực hiện B11

// Nếu DelNode có một cây con phải

B9.2: If (DelNode -> Left = NULL) and (DelNode -> Right != NULL)

B9.2.1: if (OnTheLeft = True)

PrDelNode -> Left = DelNode -> Right

B9.2.2: else

PrDelNode -> Right = DelNode -> Right

B9.2.3: DelNode -> Right = NULL

B9.2.4: Thực hiện B11

// Nếu DelNode có một cây con trái

B9.3: If (DelNode -> Left != NULL) and (DelNode -> Right = NULL)

B9.3.1: if (OnTheLeft = True)

PrDelNode -> Left = DelNode -> Left

B9.3.2: else

PrDelNode -> Right = DelNode -> Left

B9.3.3: DelNode -> Left = NULL

B9.3.4: Thực hiện B11

// Nếu DelNode có hai cây con

B10: If (DelNode -> Left != NULL) and (DelNode -> Right != NULL)

// Tìm nút trái nhất trong cây con phải của DelNode và nút cha của nó

B10.1: MLNode = DelNode -> Right

B10.2: PrMLNode = DelNode

B10.3: if (MLNode -> Left = NULL)

Thực hiện B10.7

B10.4: PrMLNode = MLNode

B10.5: MLNode = MLNode -> Left

B10.6: Lặp lại B10.3

// Chép dữ liệu từ MLNode về DelNode

B10.7: DelNode -> Key = MLNode -> Key

// Chuyển cây con phải của MLNode về cây con trái của PrMLNode

B10.8: if (PrMLNode = DelNode) // MLNode là nút phải của PrMLNode

PrMLNode -> Right = MLNode -> Right

B10.9: else *// MLNode là nút trái của PrMLNode*

PrMLNode -> Left = MLNode -> Right

B10.10: MLNode -> Right = NULL

// Chuyển vai trò của MLNode cho DelNode

B10.11: DelNode = MLNode

B10.12: Thực hiện B11

// Hủy DelNode

B11: delete DelNode

Bkt: Kết thúc

Cài đặt thuật toán xóa một nút trong cây BST bằng phương pháp thế mạng

Hàm Delete_Node_SB có prototype:

```
int Delete_Node_SB(BST_Type  &BST, Info  DelData);
```

Hàm thực hiện việc hủy nút có thành phần Key là DelData trên cây nhị phân tìm kiếm BST bằng phương pháp hủy phần tử thế mạng là phần tử trái nhất trong cây con phải của nút cần hủy (nếu nút cần hủy có hai cây con). Hàm trả về giá trị 1 nếu việc hủy thành công (có nút để hủy), trong trường hợp ngược lại hàm trả về giá trị 0 (không tồn tại nút có Key là DelData hoặc cây rỗng).

```
int Delete_Node_SB(BST_Type &BST, Info DelData)
{
    BST_Type DelNode = BST;
    BST_Type PrDelNode = NULL;
    int OnTheLeft = 0;
    while(DelNode != NULL){
        if(DelNode->Key == DelData)
            break;
        PrDelNode = DelNode;
        if(DelNode->Key > DelData){
            DelNode = DelNode->Left;
            OnTheLeft = 1;
        }
        else    // (DelNode->Key < DelData)
        {
            DelNode = DelNode->Right;
            OnTheLeft = 0;
        }
    }
    if(DelNode == NULL) // Khong co nut de huy
        return 0;
    if(PrDelNode == NULL){ // DelNode la nut goc
      if(DelNode->Left == NULL && DelNode->Right == NULL)
            BST = NULL;
      else
            if(DelNode->Left == NULL){
                                // DelNode co 1 cay con phai
                BST = BST->Right;
```

```
            DelNode->Right = NULL;
        }
        else
            if(DelNode->Right == NULL){
                        // DelNode co 1 cay con trai
                BST = BST->Left;
                DelNode->Left = NULL;
            }
}
else{   // DelNode khong la nut goc
  if(DelNode->Left==NULL && DelNode->Right==NULL)
        if(OnTheLeft == 1)
            PrDelNode->Left = NULL;
        else
            PrDelNode->Right = NULL;
  else
        if(DelNode->Left == NULL){
                        // DelNode co 1 cay con phai
            if(OnTheLeft == 1)
             PrDelNode->Left = DelNode->Right;
            else
             PrDelNode->Right = DelNode->Right;
            DelNode->Right = NULL;
        }
        else
            if(DelNode->Right == NULL){
                        // DelNode co 1 cay con trai
                if(OnTheLeft == 1)
                  PrDelNode->Left = DelNode->Left;
                else
                  PrDelNode->Right = DelNode->Left;
                DelNode->Left = NULL;
```

```
                }
    }
    // DelNode co hai cay con
    if(DelNode->Left!=NULL && DelNode->Right!=NULL)
    {
        BST_Type  MLNode = DelNode->Right;
        BST_Type  PrMLNode = DelNode;
        while(MLNode->Left != NULL){
            PrMLNode = MLNode;
            MLNode = MLNode->Left;
        }
        DelNode->Key = MLNode->Key;
        if(PrMLNode == DelNode)
            PrMLNode->Right = MLNode->Right;
        else
            PrMLNode->Left = MLNode->Right;
        MLNode->Right = NULL;
        DelNode = MLNode;
    }
    delete DelNode;
    return 1;
}
```

BÀI TẬP CHƯƠNG 4

-----ଓ✧ଌ-----

1. Xây dựng các hàm sau trên cây nhị phân:
   a. In các nút lá.
   b. In các nút nằm ở mức K.
   c. In các nút lá nằm ở mức K.
   d. In các nút con trái của cây.
   e. In các nút lá là con phải của cây.
   f. Đếm số nút bậc 2 của cây.
   g. Đếm số nút có nội dung là X.
   h. Đếm số nút nằm ở mức K.
   i. Cho biết nút có nội dung lớn nhất.
   j. Tính tổng các nút có nội dung lẻ.
   k. Tính tích các nút ở mức cao nhất.
   l. Cho biết mức nào có số nút nhiều nhất.
   m. Tính chiều dài đường đi trong.
   n. Tính chiều dài đường đi ngoài.

2. Xây dựng các hàm sau trên cây nhị phân:
   a. Cho biết mức nào có nhiều lá nhất.
   b. Cho biết trong cây nút bậc mấy là nhiều nhất.
   c. In ra các nút lá nằm ở mức thấp nhất.
   d. In ra các mức không có nút lá nào.
   e. Đếm số nút có trong cây bằng giải thuật không đệ qui.
   f. Duyệt cây theo các NLR bằng giải thuật không đệ qui.
   g. Duyệt cây theo các LNR bằng giải thuật không đệ qui.
   h. Duyệt cây theo mức bằng giải thuật đệ qui.
   i. Duyệt cây theo mức bằng giải thuật không đệ qui.
   j. Kiểm tra cây nhị phân có cân bằng không (cây nhị phân cân bằng có số nút cây con trái và số nút cây con phải hơn kém nhau nhiều nhất là 1).

3. Xây dựng các hàm sau trên cây nhị phân tìm kiếm (BST):
   a. In từ gốc đến nút K.
   b. In từ K về gốc.
   c. In từ gốc đến nút max.
   d. In từ nút min về gốc.
   e. In từ min đến max.
   f. In từ nút N đến nút M
   g. In các nút có nội dung lớn hơn K.
   h. Đếm số nút nhỏ hơn K.

4. Xây dựng các hàm sau trên cây nhị phân tìm kiếm (BST):
   a. Cho biết nội dung của nút nhỏ nhất.
   b. Trả về cây có gốc là nút có nội dung lớn nhất.
   c. Thêm 1 nút bằng giải thuật đệ qui.
   d. Tạo cây BST từ cây nhị phân đã có.
   e. Nhập cây BST.
   f. Kiểm tra cây nhị phân có phải là BST không theo hệ quả (duyệt LNR được dãy có thứ tự tăng dần).

CHƯƠNG 5: ĐỒ THỊ

-----ᘓ✧ᘑ-----

1. MỘT SỐ KHÁI NIỆM

1.1. Ánh xạ

Xét hai tập hợp X và Y, một ánh xạ đa trị từ X vào Y là một phép cho ứng với mỗi $x \in X$ một tập hợp con của Y, ký hiệu là Tx và Tx có thể là rỗng (ϕ).

Ta thường viết: $X \rightarrow Y$

$x \rightarrow Tx \subset Y$

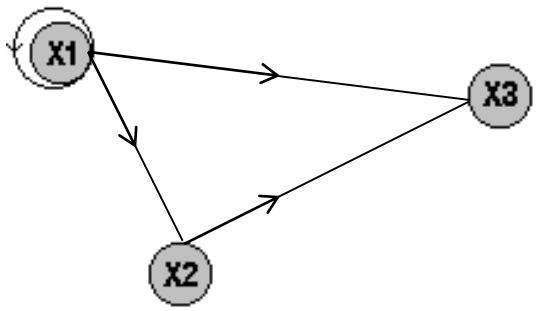
1.2. Đồ thị

Cho một đồ thị có nghĩa là cho một tập hợp V và 1 ánh xạ $T : V \rightarrow V$.

Ta ký hiệu đồ thị G = (V , T). Mỗi phần tử của V được gọi là ***đỉnh*** của đồ thị G và mỗi cặp (x , y) với $y \in Tx$ được gọi là ***cung*** của đồ thị G.

Nếu ký hiệu tập hợp cung của đồ thị G là U thì ta viết G = (T , U).

Ví dụ: Cho đồ thị (G_1) như sau:



$V = \{ x_1 , x_2 , x_3 \}$

$Tx_1 = \{ x_1 , x_2 , x_3 \}$

$Tx_2 = \{ x_3 \}$

$Tx_3 = \{ \phi \}$

$U = \{ (x_1 , x_1) , (x_1 , x_2) , (x_1 , x_3) , (x_2 , x_3) \}$

Đối với cung $u = (x , y) \in U$, ta nói x là *đỉnh gốc*, y là *đỉnh ngọn* và x , y kề nhau.

Cung (x , x) được gọi là 1 *khuyên*.

Nếu đỉnh là ngọn của một cung duy nhất và không là gốc của bất kỳ cung nào thì đỉnh này được gọi là *đỉnh treo* và cung tương ứng được gọi là *cung treo*.

Hai đồ thị được gọi là *đẳng cấu* nếu giữa các tập hợp đỉnh V và V' thiết lập một ánh xạ 1 đối 1 bảo toàn quan hệ kề nhau.

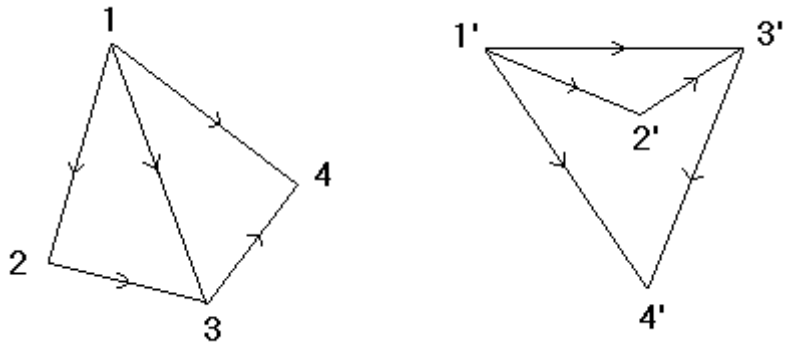
V V’

x $\longleftrightarrow$ x’

y $\longleftrightarrow$ y’

(x , y) ∈ U $\Leftrightarrow$ (x’ , y’) ∈ U’

Ví dụ: Cho 2 đồ thị (G_2) và (G_2’) như sau:



(G_2) (G_2’)

Hai đồ thị (G_2) và (G_2’) trên là đẳng cấu và ta thường viết là $G_2 \approx G_2$’

Đồ thị G’ = (V’ , T’) được gọi là *đồ thị con* của đồ thị G = (V , T) nếu V’ ⊂ V và ánh xạ T’ thỏa mãn T’x ⊂ Tx ∩ V’. Nếu V = V’ thì G’ được gọi là *đồ thị bộ phận* của G.

Đồ thị G được gọi là *đối xứng* nếu (x , y) ∈ U thì (y , x) cũng ∈ U, gọi là *phản xứng* nếu (x , y) ∈ U thì (y , x) ∉ U, gọi là *đầy đủ* nếu (x , y) ∉ U thì (y , x) phải ∈ U.

Hai cung ngược chiều nhau giữa một cặp đỉnh được thay thế bởi một đường cong hoặc một đoạn thẳng không có hướng gọi là *cạnh* của đồ thị.

Đỉnh là đầu mút của một cạnh duy nhất được gọi là *đỉnh treo*, cạnh duy nhất kề với đỉnh treo gọi là *cạnh treo*.

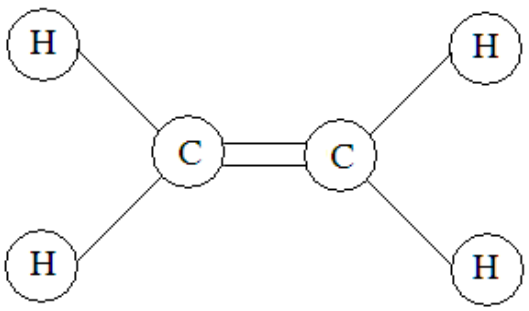
Một đồ thị đối xứng gọi là *đồ thị vô hướng* và một đồ thị không đối xứng gọi là *đồ thị hữu hướng*.

Một đồ thị G = (V , U) được gọi là *đa đồ thị* nếu nhiều cung (cạnh) có thể tương ứng với một cặp đỉnh, tức là 2 đỉnh của G có thể được nối liền nhau bởi nhiều cung.

Đối với đa đồ thị G = (V , U) trong đó V = { 1 , 2 , … , n } ta đặt tương ứng một ma trận A = (a_{ij}) với a_{ij} là số cung từ đỉnh i đến đỉnh j. Ma trận A được gọi là *ma trận kề* của G.

Nếu ma trận kề A gồm toàn những phần tử 0 và 1 thì G được gọi là *đồ thị đơn*.

Ví dụ: Đồ thị (G_3) bên đây là một đa đồ thị



✧ *Như vậy*: G đối xứng $\Leftrightarrow a_{ij} = a_{ji}$

G phản xứng $\Leftrightarrow a_{ij} + a_{ji} \leq 1$

G đầy đủ $\Leftrightarrow a_{ij} + a_{ji} \geq 1$

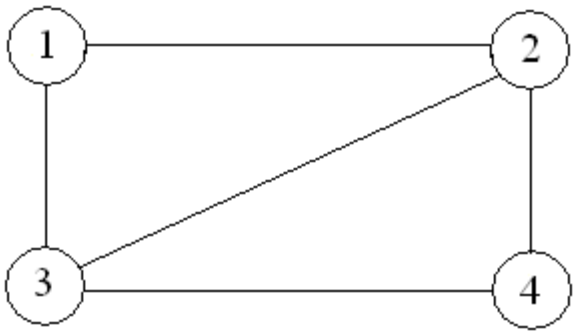
1.3. Đường đi và dây chuyền

Một đường đi (path) trên đồ thị là một dãy tuần tự các đỉnh v_1, v_2, ..., v_n sao cho (v_i, v_{i+1}) là một cạnh trên đồ thị (i = 1, 2, ..., n-1). Đường đi này là đường đi từ v_1 đến v_n và đi qua các đỉnh v_2, ..., v_{n-1}. Đỉnh v_1 gọi là đỉnh đầu, v_n gọi là đỉnh cuối. Độ dài của đường đi này bằng (n - 1). Trường hợp đặc biệt dãy chỉ có một đỉnh v thì ta coi đó là đường đi từ v đến chính nó có độ dài bằng không.

Ví dụ: Dãy 1, 2, 4 trong đồ thị (G_4) là một đường đi từ đỉnh 1 đến đỉnh 4, đường đi này có độ dài là hai.

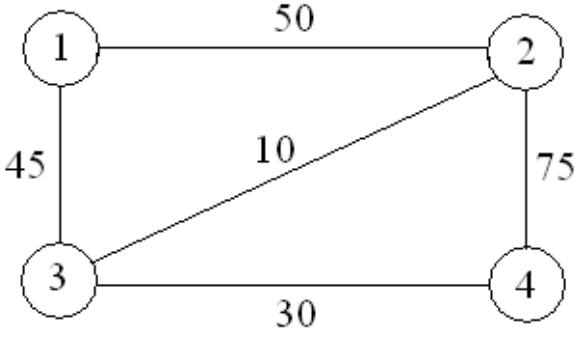
Đường đi gọi là đơn (simple) nếu mọi đỉnh trên đường đi đều khác nhau, ngoại trừ đỉnh đầu và đỉnh cuối có thể trùng nhau. Một đường đi có đỉnh đầu và đỉnh cuối trùng nhau gọi là một chu trình (cycle). Một chu trình đơn là một đường đi đơn có đỉnh đầu và đỉnh cuối trùng nhau và có độ dài ít nhất là 1.

Ví dụ: Cho đồ thị (G_4) như sau:



Trong đồ thị (G_4) thì 1, 3, 2, 1 tạo thành một chu trình có độ dài 3 và 1, 3, 4, 2, 1 là một chu trình có độ dài 4.

Trong nhiều ứng dụng ta thường kết hợp các giá trị (value) hay nhãn (label) với các đỉnh và/hoặc các cạnh, lúc này ta nói đồ thị có nhãn. Nhãn kết hợp với các đỉnh và/hoặc cạnh có thể biểu diễn tên, giá, khoảng cách,... Nói chung nhãn có thể có kiểu tuỳ ý. Đồ thị (G_5) sau đây cho ta ví dụ về một đồ thị có nhãn. Ở đây nhãn là các giá trị số nguyên biểu diễn cho giá cước vận chuyển một tấn hàng giữa các thành phố 1, 2, 3, 4 chẳng hạn.



Một dãy đỉnh và cung liên tiếp nhau trên đồ thị G không đối xứng (hữu hướng) được gọi là *dây chuyền*. Một dây chuyền mà điểm cuối của mỗi cung (trừ cung cuối) là điểm đầu của cung kế tiếp được gọi là *mạch*. Nếu đỉnh đầu và đỉnh cuối trùng nhau thì ta có *dây chuyền (mạch) đóng.* Một đường đi (dây chuyền), một chu trình (dây chuyền đóng, mạch đóng) được gọi là *sơ cấp* nếu nó không đi qua đỉnh nào quá 1 lần, và gọi là *đơn giản* nếu nó không chứa cạnh (cung) nào quá 1 lần.

1.4. Tính liên thông

Một đồ thị hữu hướng G được gọi là *liên thông* nếu với mọi cặp đỉnh phân biệt bao giờ cũng có một dây chuyền từ đỉnh này đến đỉnh kia, và được gọi là *liên thông mạnh* nếu với bất kỳ cặp đỉnh X và Y nào ($X \neq Y$) bao giờ cũng có một mạch từ X đến Y và ngược lại.

Trong đồ thị đối xứng (vô hướng), hai đỉnh X và Y được gọi là *liên thông* nếu chúng được nối liền với nhau bởi ít nhất một đường đi, ngược lại thì chúng được gọi là *không liên thông*.

Gọi Cx là tập các đỉnh liên thông với đỉnh x (kể cả x), đồ thị con của G được tạo ra bởi tập đỉnh Cx được gọi là *thành phần liên thông* của G và ký hiệu là Gx.

✧ Đồ thị G liên thông khi và chỉ khi nó gồm một thành phần liên thông duy nhất.

2. PHÂN LOẠI ĐỒ THỊ

Chúng ta có thể phân loại đồ thị bằng nhiều tiêu chí khác nhau. Tùy theo mục đích sử dụng đồ thị mà ta dựa vào tiêu chí thích hợp để phân loại chúng.

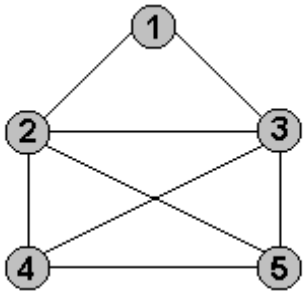
- Đồ thị vô hướng – Đồ thị hữu hướng
- Đồ thị đầy – Đồ thị thưa
- Đồ thị liên thông – Đồ thị không liên thông
- Đơn đồ thị – Đa đồ thị

Phân biệt đồ thị đầy – đồ thị thưa dùng để quyết định chọn cấu trúc dữ liệu nào để cài đặt cho đồ thị.

3. ĐỒ THỊ ĐẦY

3.1. Định nghĩa

Đồ thị đầy (dense) là đồ thị có số phần tử là 1 nhiều hơn số phần tử là 0 trong ma trận kề của nó. Giả sử ta có đồ thị vô hướng (G_6) như sau:



3.2. Ma trận kề

$$\begin{bmatrix} 1 & 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 & 1 \end{bmatrix}$$

⇨ Đồ thị (G_6) là đồ thị đầy. Thông thường đồ thị vô hướng là đồ thị đầy.

Người ta sử dụng ma trận kề để lưu trữ đồ thị đầy, chính là mảng 2 chiều (ma trận vuông cấp N, với N là số đỉnh của đồ thị). Nếu đỉnh x có cạnh nối với đỉnh y thì gán giá trị ***1*** cho phần tử [x , y] và [y , x], ngược lại thì gán ***0*** cho phần tử [x , y] và [y , x] và ta gán **1** cho các phần tử [x , x] $\forall$ x $\in$ V.

Để đơn giản cho việc viết chương trình ta qui ước bỏ hàng và cột đầu tiên trong khai báo (hàng 0 và cột 0). Theo qui ước này thì đỉnh thứ i của đồ thị sẽ được lưu trữ ở hàng i và cột i trong ma trận. Như vậy đồ thị (G_6) được lưu trong bộ nhớ có dạng như sau:

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

Gọi: **maxn** là số đỉnh tối đa của đồ thị.

g là ma trận vuông biểu diễn đồ thị.

n là số đỉnh của đồ thị.

e số cạnh của đồ thị.

3.3. Khai báo cấu trúc dữ liệu

```
const int maxn = 20;
typedef int Graph[maxn][maxn];
```

3.4. Khởi tạo đồ thị rỗng

```
void Initialize(Graph &g, int n, int &e)
{
    int i, j;
    for(i=1; i<=n; i++)
      for(j=1; j<=n; j++)
          if (i==j) g[i][j]=1;
```

```
            else g[i][j]=0;
    e=0;
}
```

3.5. Thêm một cạnh

```
void InsertEdge(Graph &g, int n, int &e, int d1, int d2)
{
    g[d1][d2]=1;
    g[d2][d1]=1;
    e++;
}
```

3.6. Xóa một cạnh

```
void DeleteEdge(Graph &g, int n, int &e, int d1, int d2)
{
   g[d1][d2]=0;
    g[d2][d1]=0;
    e--;
}
```

3.7. Đếm số cạnh nối của một đỉnh

```
int CountEdge(Graph g, int n, int e, int d)
{   int i, dem=0;
    for(i=1; i<=n; i++)
        if(g[d][i]==1) dem++;
    return dem-1;
}
```

3.8. Thêm một đỉnh

```
void InsertVertice(Graph &g, int &n, int &e, int sc)
{   int i, d;
    for(i=1; i<=n; i++)
    {
        g[i][n+1]=0;
        g[n+1][i]=0;
    }
```

```
    g[n+1][n+1]=1;
    for(i=1; i<=sc; i++)
    {
        cout<<"Nhap dinh noi voi dinh moi : ";
        cin>>d;
        InsertEdge(g, n, e, n+1, d);
    }
    n++;
}
```

3.9. Xóa một đỉnh

```
void DeleteVertice(Graph &g, int &n, int &e, int d)
{
    int i, j;
    e=e-CountEdge(g, n, e, d);
    for(i=d; i<=n-1; i++)
        for(j=1; j<=n; j++)
            g[i][j]=g[i+1][j];
    for(i=1; i<=n-1; i++)
        for(j=d; j<=n-1; j++)
            g[i][j]=g[i][j+1];
    for (i=1; i<=n; i++)
    {
        g[i][n]=0;
        g[n][i]=0;
    }
    n--;
}
```

3.10. Nhập đồ thị

```
void ReadGraph(Graph &g, int n, int e)
{
    int i, d1, d2;
```

```
    for(i=1; i<=e; i++){
        cout<<"Nhap dinh dau va dinh cuoi : ";
        cin>>d1>>d2;
        g[d1][d2]=1;
        g[d2][d1]=1;
    }
}
```

3.11. Xuất đồ thị

```
void PrintGraph(Graph g, int n, int e)
{
    int i, j;
    for(i=1; i<=n; i++){
        for(j=1; j<=n; j++)
            cout<<g[i][j]<<'\t';
        cout<<endl;
    }
}
```

3.12. Hàm main( )

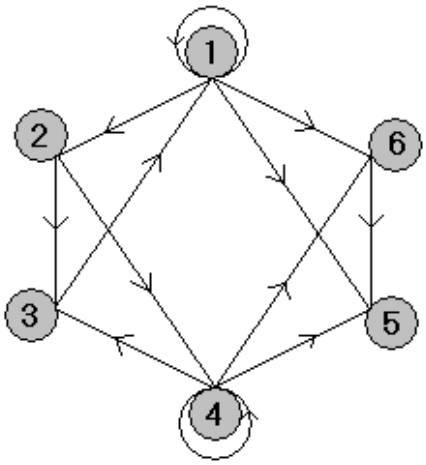
```
main()
{
    Graph g;
    int n, e, d1, d2, d, sc;
    do{
        cout<<"Nhap so dinh cua do thi : ";
        cin>>n;
    }while(n<=0 || n>=maxn);
    Initialize(g, n, e);
    cout<<"=====DO THI VUA KHOI TAO====="<<endl;
    PrintGraph(g, n, e);
    cout<<"Nhap so canh cua do thi : ";
    cin>>e;
    ReadGraph(g, n, e);
```

```
    cout<<"=====DO THI VUA NHAP=====<<endl;
    PrintGraph(g, n, e);
    cout<<endl;
    cout<<"Nhap canh can them : ";
    cin>>d1>>d2;
    InsertEdge(g, n, e, d1, d2);
    cout<<"=====DO THI SAU KHI THEM CANH=====<<endl;
    PrintGraph(g, n, e);
    cout<<endl;
    cout<<"Nhap canh can xoa : ";
    cin>>d1>>d2;
    DeleteEdge(g, n, e, d1, d2);
    cout<<"=====DO THI SAU KHI XOA CANH=====<<endl;
    PrintGraph(g, n, e);
    cout<<endl;
    cout<<"Nhap dinh can dem so canh : ";
    cin>>d;
    cout<<"Dinh "<<d<<" co "<<CountEdge(g, n, e, d);
    cout<<" canh noi"<<endl;
    cout<<"Nhap so canh noi cua dinh moi : ";
    cin>>sc;
    InsertVertice(g, n, e, sc);
    cout<<"=====DO THI SAU KHI THEM DINH=====<<endl;
    PrintGraph(g, n, e);
    cout<<endl;
    cout<<"Nhap dinh can xoa : ";
    cin>>d;
    DeleteVertice(g, n, e, d);
    cout<<"=====DO THI SAU KHI XOA DINH=====<<endl;
    PrintGraph(g, n, e);
}
```

4. ĐỒ THỊ THƯA

4.1. Định nghĩa

Đồ thị thưa (sparse) là đồ thị có số phần tử là 1 ít hơn số phần tử là 0 trong ma trận kề của nó. Giả sử ta có đồ thị (G_7) như sau:



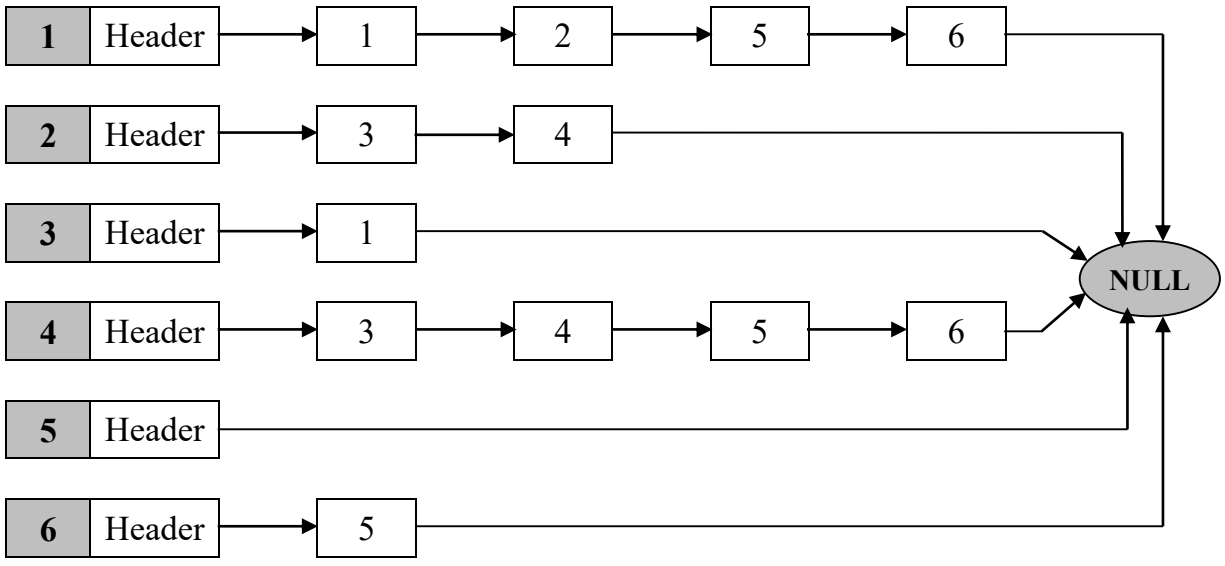
4.2. Ma trận kề

$$\begin{bmatrix} 1 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix}$$

⇨ Đồ thị (G_7) là đồ thị thưa. Thông thường đồ thị hữu hướng là đồ thị thưa.

Người ta sử dụng danh sách kề để lưu trữ đồ thị thưa, chính là mảng một chiều mà mỗi phần tử của nó là một danh sách liên kết. Nếu đỉnh X có cung (đường đi) đến đỉnh Y thì lưu Y vào danh sách liên kết thứ X trong mảng.

Đồ thị (G_7) được lưu trong bộ nhớ có dạng như sau:



Gọi: **maxn** là số đỉnh tối đa của đồ thị.

g là mảng 1 chiều biểu diễn đồ thị.

n là số đỉnh của đồ thị.

e số cung của đồ thị.

4.3. Khai báo cấu trúc dữ liệu

```
const int maxn = 20;
typedef int ElementType;
struct Node
{
    ElementType Element;
    Node* Next;
};
typedef Node* List;
typedef  List Graph[maxn];
```

4.4. Khởi tạo đồ thị rỗng

```
void Initialize(Graph &g, int n, int &e)
{
    for(int i=1; i<=n; i++)
    {
        g[i] = new Node;
        g[i]->Next = NULL;
    }
    e=0;
}
```

4.5. Thêm một cung

```
void InsertArc(Graph &g, int n, int &e, int d1, int d2)
{
    Node *t = new Node;
    t->Element = d2;
    t->Next = g[d1]->Next;
    g[d1]->Next = t;
    e++;
}
```

4.6. Xóa một cung

```
void DeleteArc(Graph &g, int n, int &e, int d1, int d2)
{   Node *t = g[d1];
    while(t->Next != NULL && t->Next->Element != d2)
        t = t->Next;
    if(t->Next != NULL && t->Next->Element == d2)
    {   Node *p = t->Next;
        t->Next = p->Next;
        delete p;
        e--;
    }
}
```

4.7. Đếm số cung xuất phát từ một đỉnh

```
int CountArcFromVertice(Graph g, int n, int e, int d)
{   int dem = 0;
    Node *t = g[d];
    while(t->Next != NULL)
    {
        dem++;
        t = t->Next;
    }
    return dem;
}
```

4.8. Đếm số cung đi đến một đỉnh

```
int CountArcToVertice(Graph g, int n, int e, int d)
{   int i, dem = 0;
    Node *t;
    for(i=1; i<=n; i++)
    {
        t = g[i];
        while(t->Next != NULL && t->Next->Element != d)
          t = t->Next;
```

```
        if(t->Next != NULL) dem++;
    }
    return dem;
}
```

4.9. Thêm một đỉnh

```
void InsertVertice(Graph &g, int &n, int &e,
                                    int scxp, int scdd)
{   int i, d;
    g[n+1]= new Node;
    g[n+1]->Next = NULL;
    for(i=1; i<=scxp; i++)
    {
        cout<<"Dinh moi di den dinh nao? ";
        cin>>d;
        InsertArc(g, n, e, n+1, d);
    }
    for(i=1; i<=scdd; i++)
    {
        cout<<"Dinh nao di den dinh moi? ";
        cin>>d;
        InsertArc(g, n, e, d, n+1);
    }
    n++;
}
```

4.10. Xóa một đỉnh

```
void DeleteVertice(Graph &g, int &n, int &e, int d)
{   int i;
    e=e-CountArcFromVertice(g, n, e, d);
    for(i=1; i<=n; i++)
        if(i==d) g[i]->Next = NULL;
        else DeleteArc(g, n, e, i, d);
    for(i=d; i<= n-1; i++)
```

```
        g[i] = g[i+1];
    n--;
    for(i=1; i<=n; i++)
    {
        Node *t = g[i];
        while(t->Next != NULL)
        {
            if(t->Next->Element > d)
                t->Next->Element = t->Next->Element - 1;
            t = t->Next;
        }
    }
}
```

4.11. Nhập đồ thị

```
void ReadGraph(Graph &g, int n, int e)
{
    int i, d1, d2, sc = e;
    for(i=1; i<=sc; i++)
    {
        cout<<"Nhap cung di tu d1 den d2 : ";
        cin>>d1>>d2;
        InsertArc(g, n, e, d1, d2);
    }
}
```

4.12. Xuất đồ thị

```
void PrintGraph(Graph g, int n, int e)
{
    Node *t = new Node;
    for(int i=1; i<=n; i++)
    {
        cout<<"Dinh "<<i<<" : ";
        t = g[i];
        while(t->Next != NULL)
```

```
        {
            cout<<t->Next->Element<<'\t';
            t = t->Next;
        }
        cout<<endl;
    }
}
```

4.13. Hàm main( )

```
main()
{   Graph g;
    int n, e, d1, d2, d, scxp, scdd;
    do{
         cout<<"Nhap so dinh cua do thi : ";
         cin>>n;
    }while(n<=0 || n>=maxn);
    Initialize(g, n, e);
    cout<<"=====DO THI VUA KHOI TAO LA====="<<endl;
    PrintGraph(g, n, e);
    cout<<"Nhap so cung cua do thi : "; cin>>e;
    ReadGraph(g, n, e);
    cout<<"=====DO THI VUA NHAP LA====="<<endl;
    PrintGraph(g, n, e);
    cout<<endl;
    cout<<"Nhap cung can them : "; cin>>d1>>d2;
    InsertArc(g, n, e, d1, d2);
    cout<<"=====DO THI SAU KHI THEM CUNG LA====="<<endl;
    PrintGraph(g, n, e);
    cout<<endl;
    cout<<"Nhap cung can xoa : "; cin>>d1>>d2;
    DeleteArc(g, n, e, d1, d2);
    cout<<"=====DO THI SAU KHI XOA CUNG LA====="<<endl;
    PrintGraph(g, n, e);
    cout<<endl;
```

```
    cout<<"Nhap dinh can dem so cung : "; cin>>d;
    cout<<"Dinh "<<d<<" co "<<CountArcFromVertice(g,n,e,d);
    cout<<" cung xuat phat"<<endl;
    cout<<"Dinh "<<d<<" co "<<CountArcToVertice(g,n,e,d);
    cout<<" cung den"<<endl;
     cout<<"Nhap so cung xuat tu dinh moi : ";
     cin>>scxp;
     cout<<"Nhap so cung di den dinh moi : ";
     cin>>scdd;
     InsertVertice(g, n, e, scxp, scdd);
     cout<<"=====DO THI SAU KHI THEM DINH LA====="<<endl;
     PrintGraph(g, n, e);
     cout<<endl;
     cout<<"Nhap dinh can xoa : "; cin>>d;
     DeleteVertice(g, n, e, d);
     cout<<"=====DO THI SAU KHI XOA DINH LA====="<<endl;
     PrintGraph(g, n, e);
}
```

BÀI TẬP CHƯƠNG 5

-----ଓ✧ଌ-----

1. Cho đồ thị vô hướng đầy, xây dựng các hàm sau:
   a. Cho biết đỉnh nào có số cạnh nối là K.
   b. Nhận vào 1 đỉnh, in ra các đỉnh có cạnh nối với đỉnh này.
   c. Nhận vào 2 đỉnh, cho biết giữa chúng có cạnh nối không?
   d. Cho biết có bao nhiêu đỉnh treo trong đồ thị.
   e. In ra các đỉnh có cạnh nối với các đỉnh treo.
   f. In ra đỉnh có cạnh nối nhiều nhất.
   g. Kiểm tra ma trận bất kỳ có phải là ma trận kề của 1 đồ thị nào không?

2. Cho đồ thị hữu hướng thưa, xây dựng các hàm sau:
   a. Cho biết đỉnh có số cung đến các đỉnh khác nhiều nhất.
   b. Cho biết đỉnh có số cung đến từ các đỉnh khác ít nhất.
   c. Đếm số khuyên có trong đồ thị.
   d. Cho biết có bao nhiêu đỉnh treo trong đồ thị.
   e. In ra các đỉnh có cung đến đỉnh d.
   f. In ra các cặp đỉnh có tổng số cung xuất phát và cung đến bằng nhau.
   g. Đếm xem có bao nhiêu đỉnh có số cung xuất phát bằng số cung đến.

3. Cài đặt đồ thị hữu hướng đầy:
   a. Khai báo cấu trúc dữ liệu.
   b. Thêm 1 cung.
   c. Xóa 1 cung.
   d. Đếm số cung xuất phát từ 1 đỉnh.
   e. Đếm số cung đến 1 đỉnh.
   f. Thêm 1 đỉnh.
   g. Xóa 1 đỉnh.
   h. Nhập đồ thị.
   i. Xuất đồ thị.

4. Cài đặt đồ thị vô hướng thưa:
   a. Khai báo cấu trúc dữ liệu.
   b. Thêm 1 cạnh.
   c. Xóa 1 cạnh.
   d. Đếm số cạnh nối của 1 đỉnh.
   e. Thêm 1 đỉnh.
   f. Xóa 1 đỉnh.
   g. Nhập đồ thị.
   h. Xuất đồ thị.

CHƯƠNG 6: BẢNG BĂM

-----ଓ✧ଌ-----

1. TỔNG QUAN

1.1. Phép băm

Phép băm (phép biến đổi khóa) là một phương pháp tham khảo trực tiếp các phần tử trong một bảng thông qua việc biến đổi số học trên những khóa để được địa chỉ tương ứng của những phần tử trong bảng.

Tổng quát: Phép băm là 1 ánh xạ thích hợp H từ tập các khóa K vào tập các địa chỉ A.
$H : K \rightarrow A$

Trong phép băm thông thường tập các khóa lớn hơn nhiều so với tập các địa chỉ bộ nhớ (chỉ số của mảng).

Như vậy hàm H là hàm Nhiều - Một.

Để thực hiện băm ta có 2 bước:

- *Bước 1*: Tính toán hàm H để biến đổi khóa cần tìm thành địa chỉ trong bảng.
- *Bước 2*: Giải quyết sự đụng độ cho những khóa khác nhau lại có cùng địa chỉ trong bảng.

Cấu trúc bảng băm đơn giản, người ta có thể sử dụng danh sách liên kết hoặc danh sách đặc để giải quyết sự đụng độ, danh sách liên kết được sử dụng nhiều hơn vì ta không biết trước số các khóa khác nhau có cùng địa chỉ trong bảng là bao nhiêu.

1.2. Ứng dụng của bảng băm

Bảng băm được đề xuất và hiện thực trên máy tính từ những năm 50 của thế kỷ 20. Nó dựa trên ý tưởng là chuyển đổi khóa thành một số (xử lý băm) và sử dụng số này để đánh chỉ số cho bảng dữ liệu.

Các phép toán trên các cấu trúc dữ liệu như danh sách liên kết, cây nhị phân,… phần lớn được thực hiện bằng cách so sánh các phần tử của cấu trúc, vì thế thời gian truy xuất không nhanh và phụ thuộc vào kích thước của cấu trúc. Chương này sẽ khảo sát một cấu trúc dữ liệu mới được gọi là bảng băm (hash table). Các phép toán trên bảng băm sẽ giúp hạn chế số lần so sánh và do đó sẽ giảm thiểu được thời gian truy xuất. Độ phức tạp của các phép toán trên bảng băm thường có bậc là O(1) và không phụ thuộc vào kích thước của bảng băm.

Có nhiều loại bảng băm như: bảng băm chữ nhật (m hàng x n cột), bảng băm tam giác dưới (m hàng), bảng băm tam giác trên (n cột), bảng băm đường chéo và bảng băm ADT,… Trong đó bảng băm ADT được sử dụng nhiều nhất.

2. HÀM BĂM (HASH FUNCTION)

2.1. Định nghĩa

Hàm băm dùng để biến đổi các khóa thành các địa chỉ trong bảng. Yêu cầu của hàm băm là khả năng phân bố đều trên miền giá trị của địa chỉ. Hàm băm còn được gọi là "*hàm biến đổi khóa*".

Gọi M là số phần tử được chứa trong bộ nhớ (số phần tử của bảng), hàm băm sẽ biến đổi các khóa thành các số nguyên trong đoạn [0 .. M - 1].

Giả sử khóa là các số nguyên, ta có thể sử dụng hàm HF(k) là: HF(k) = k % M, với k là khóa và HF(k) là hàm lấy số dư của phép chia k cho M.

Ta thường chọn M là số nguyên tố vì đặc tính số học của phép toán % (chia lấy phần dư).

2.2. Ví dụ

Ta có các khóa như sau:

8 13 6 5 9 3 15 11 12

Chọn M = 5

⇨ Địa chỉ của các khóa trong bảng là:

Khóa:	8	13	6	5	9	3	15	11	12
Địa chỉ:	3	3	1	0	4	3	0	1	2

Ta thấy có nhiều khóa khác nhau nhưng lại có cùng địa chỉ trong bảng (sự đụng độ).

Như vậy hàm băm chính là hàm Nhiều - Một.

2.3. Phân loại

Trong hầu hết các ứng dụng, khóa được dùng như một phương thức để truy xuất dữ liệu một cách gián tiếp. Khóa có thể là dạng số hay dạng chuỗi. Tùy theo kiểu dữ liệu của khóa mà ta có thể xây dựng các hàm băm (hàm biến đổi khóa) khác nhau.

Một hàm băm được gọi là tốt nếu nó thỏa mãn các điều kiện sau:

- Tính toán nhanh
- Các khóa được phân bố đều trong bảng
- Ít xảy ra đụng độ

Nếu khóa không phải là số nguyên thì ta có thể giải quyết như sau:

- *Bước 1:* Tìm cách biến đổi khóa thành số nguyên
- *Bước 2:* Sử dụng các hàm băm chuẩn trên số nguyên

Ví dụ 1: Nếu khóa là số thực thì ta có thể sử dụng hàm làm tròn để biến đổi nó thành số nguyên.

Ví dụ 2: Nếu khóa có dạng mã số như 6475-7661 thì ta loại bỏ dấu "-" để đưa nó về số nguyên 64757661.

Ví dụ 3: Nếu khóa là chuỗi thì ta có thể sử dụng bảng mã ASCII để biến đổi nó thành số nguyên tương ứng.

2.3.1 Hàm băm dùng cho khóa là số nguyên

Hàm băm chỉ đơn giản là: HF(k) = k % M

Trong đó:

- k là khóa
- M là kích thước của bảng, thông thường M được chọn là số ***nguyên tố***

2.3.2. Hàm băm dùng cho khóa là chuỗi

Thuật toán Horner tổng quát

Đối với các khóa là chuỗi thì ta phải tính hàm băm sao cho không bị tràn số. Trong trường hợp này ta dùng thuật toán Horner để tính giá trị của hàm băm.

Thuật toán Horner tính hàm băm như sau:

```
t = key[ 0 ] % M;
for  (i= 1;  i < keysize; i++)
    t = ( t * < cơ số > + key[ i ] ) % M;
HF = t;
```

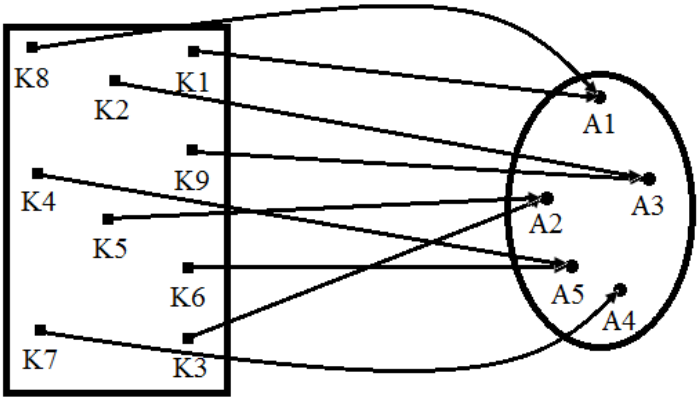
Hàm băm dựa vào thuật toán Horner và bảng mã ASCII:

```
int Hash_ASCII(char* s)
{   int i, t;
    t = s[0] % M;
    for  (i = 1; i < strlen(s); i++)
        t = ( t * 10 + s[i]) % M;
    return t;
}
```

3. BẢNG BĂM ADT

3.1. Mô tả

Giả sử ta có: K là tập các khóa, M là tập các địa chỉ trong bảng và HF(k) là hàm băm dùng để ánh xạ một khóa k từ tập các khóa K thành một địa chỉ tương ứng trong tập M.



3.2. Các phép toán

3.2.1 Khởi tạo (Initialize)

Khởi tạo bảng băm là cấp phát vùng nhớ hay qui định số phần tử (kích thước) của bảng băm.

3.2.2 Kiểm tra rỗng (Empty)

Kiểm tra bảng băm có rỗng hay không.

3.2.3. Lấy kích thước của bảng (Size)

Cho biết số phần tử hiện có trong bảng.

3.2.4. Tìm kiếm (Search)

Tìm kiếm một phần tử trong bảng băm theo khóa k chỉ định trước.

3.2.5. Thêm mới phần tử (Insert)

Thêm một phần tử vào bảng băm, sau khi thêm số phần tử của bảng tăng 1.

3.2.6. Loại bỏ (Remove)

Loại bỏ một phần tử ra khỏi bảng băm, sau khi loại bỏ số phần tử của bảng giảm 1.

3.2.7. Sao chép (Copy)

Tạo một bảng băm mới từ một bảng băm đã có.

3.2.8. Duyệt (Traverse)

Duyệt bảng băm theo thứ tự địa chỉ từ nhỏ đến lớn.

3.3. Các bảng băm thông dụng

Với mỗi loại bảng băm cần thiết phải xác định tập khóa K, xác định tập địa chỉ M và xây dựng hàm băm HF cho phù hợp.

Mặt khác, khi xây dựng hàm băm cũng cần thiết phải tìm kiếm các giải pháp để giải quyết sự đụng độ, nghĩa là làm giảm thiểu sự ánh xạ của nhiều khóa khác nhau vào cùng một địa chỉ trong bảng.

3.3.1 Bảng băm với phương pháp nối kết trực tiếp

Mỗi địa chỉ của bảng băm (gọi là một bucket) tương ứng với một danh sách liên kết.

Các phần tử bị đụng độ được nối kết với nhau trên một danh sách liên kết.

3.3.2. Bảng băm với phương pháp nối kết hợp nhất

Bảng băm loại này được cài đặt bằng danh sách đặc, mỗi phần tử có 2 trường: trường *key* chứa khóa của phần tử và trường *Next* chỉ đến phần từ kế bị đụng độ.

Các phần tử bị đụng độ được nối kết với nhau qua trường Next.

3.3.3. Bảng băm với phương pháp thử tuyến tính

Khi thêm phần tử vào bảng băm loại này nếu băm lần đầu bị đụng độ thì lần lượt thử (dò tìm) địa chỉ kế. Thử cho đến khi gặp địa chỉ trống đầu tiên thì thêm phần tử vào địa chỉ này.

3.3.4. Bảng băm với phương pháp thử bậc hai

Tương tự như bảng băm với phương pháp thử tuyến tính, khi thêm phần tử vào bảng băm loại này nếu băm lần đầu bị đụng độ thì lần lượt thử (dò tìm) đến địa chỉ mới, lần thử thứ i ở vị trí cách khoảng i^2. Thử cho đến khi gặp địa chỉ trống đầu tiên thì thêm phần tử vào địa chỉ này.

3.3.5. Bảng băm với phương pháp băm kép

Bảng băm loại này dùng 2 hàm băm khác nhau, băm lần đầu với hàm băm thứ nhất nếu bị đụng độ thì tìm địa chỉ khác bằng hàm băm thứ hai.

3.4. Ưu điểm của bảng băm

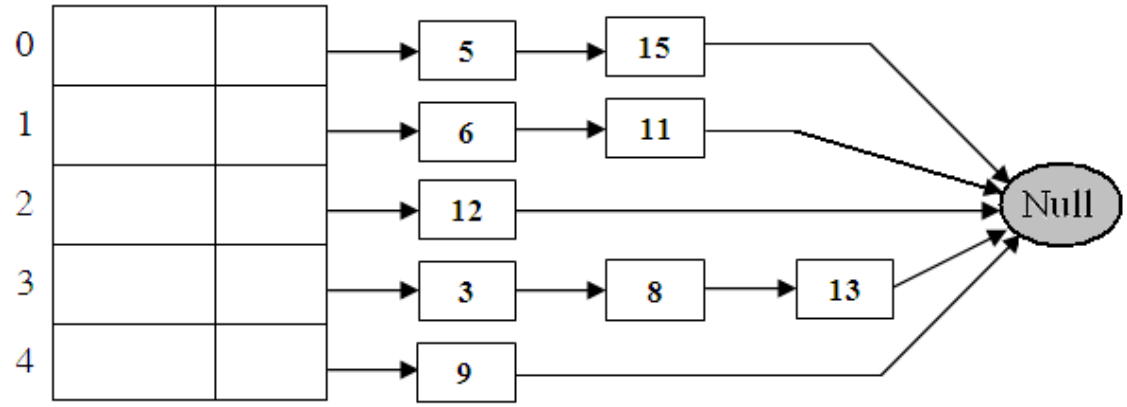
- Là một cấu trúc dung hòa giữa thời gian truy xuất và dung lượng bộ nhớ.
- Nếu không có sự giới hạn về bộ nhớ thì chúng ta có thể xây dựng bảng băm với mỗi khóa ứng với một địa chỉ với mong muốn thời gian truy xuất tức thời.
- Nếu dung lượng bộ nhớ có giới hạn thì tổ chức một số khóa có cùng địa chỉ, lúc này thời gian truy xuất bị suy giảm đôi chút.
- Được ứng dụng nhiều trong thực tế, rất thích hợp khi tổ chức dữ liệu có kích thước lớn và được lưu trữ ở bộ nhớ ngoài.

4. GIẢI QUYẾT SỰ ĐỤNG ĐỘ BẰNG PHƯƠNG PHÁP NỐI KẾT

4.1. Phương pháp nối kết trực tiếp

Ứng với mỗi địa chỉ của bảng là một danh sách liên kết chứa các phần tử có khóa khác nhau nhưng lại có cùng địa chỉ. Như vậy, ta cần phải sử dụng mảng một chiều gồm M phần tử, với mỗi phần tử có một con trỏ là chỉ điểm đầu của mỗi danh sách liên kết tương ứng. Các phần tử trong bảng được "băm" thành M danh sách liên kết (từ danh sách 0 đến danh sách M-1). Các phần tử bị đụng độ tại địa chỉ i sẽ được nối kết trực tiếp với nhau qua danh sách liên kết i. Chẳng hạn với M = 5, các phần tử có hàng đơn vị là 0 hoặc 5 sẽ được băm vào danh sách liên kết đầu tiên trong bảng (i = 0).

4.1.1 Cách biểu diễn



4.1.2. Khai báo cấu trúc dữ liệu

```
# define M 5
struct Node
{    int Key;
     Node* Next;
};
// Khai bao kieu con tro chi danh sach lien ket
typedef Node* Bucket;
// Khai bao mang bucket chua M con tro dau cua M bucket
Bucket Hash[M];
```

4.1.3 Các phép toán

Hàm băm

Giả sử ta sử dụng hàm băm dạng %: HF(k) = k % M.

```
int HF(int k)
{
     return (k % M);
}
```

✧ Ta cũng có thể dùng một hàm băm bất kỳ thay cho hàm băm dạng % trên.

Khởi tạo

```
void Initialize()
{    for(int b = 0; b < M; b++)
     {
          Hash[b] = new Node;
          Hash[b]->Next = NULL;
     }
}
```

Kiểm tra một bucket rỗng

```
bool Empty_Bucket(int b)
{
     return(Hash[b]->Next == NULL);
}
```

Kiểm tra bảng băm rỗng

```
bool Empty()
{    for(int b=0; b < M; b++)
          if(!Empty_Bucket(b))
               return false;
     return true;
}
```

Tìm một khóa trong bảng băm

Tìm khóa k trong bảng băm, nếu tìm thấy hàm sẽ trả về giá trị là true, ngược lại hàm sẽ trả về giá trị là false.

```
bool Search(int k)
{    Bucket p;
     int b = HF(k);
     p = Hash[b];
```

```
     while(p->Next != NULL && p->Next->Key < k)
          p = p->Next;
     if(p->Next != NULL && p->Next->Key == k)
          return true;
     else
          return false;
}
```

Thêm một khóa vào bucket

Giả sử ta cần thêm khóa k vào bucket b trong bảng băm, vì các khóa trong bucket có thứ tự tăng dần nên giải thuật tìm vị trí đúng cho khóa trong bucket phải áp dụng tính chất này.

```
void Insert_Bucket(int b, int k)
{    Bucket t = new Node;
     t->Key = k;
     if(Empty_Bucket(b) || k < Hash[b]->Next->Key)
     {
          t->Next = Hash[b]->Next;
          Hash[b]->Next = t;
     }
     else
     {
          Bucket p = Hash[b];
          while(p->Next != NULL && p->Next->Key < k)
               p = p->Next;
          t->Next = p->Next;
          p->Next = t;
     }
}
```

Thêm một khóa vào bảng băm

Dựa vào giải thuật thêm một khóa vào bucket ta có thể dễ dàng thêm một khóa vào bảng băm bằng cách xác định xem khóa đó cần đưa vào bucket nào của bảng băm. Ta cũng lưu ý trường hợp trong bảng băm không có các khóa trùng nhau.

```
void Insert(int k)
{
     int b = HF(k);
     if(!Search(k)) Insert_Bucket(b, k);
}
```

Xóa một khóa khỏi bảng băm

Để thực hiện việc xóa một khóa ra khỏi bảng băm trước tiên chúng ta cần xác định bucket phù hợp, sau đó tìm khóa cần xóa trong bucket này, nếu tìm thấy thì xóa chúng ra khỏi bucket.

```
void Remove(int k)
{
     int b = HF(k);
     Bucket p = Hash[b];
     while(p->Next != NULL && p->Next->Key < k)
          p = p->Next;
     if(p->Next == NULL || p->Next->Key != k)
         cout<<"Khong co khoa nay"<<endl;
     else
     {
          Bucket t = p->Next;
          p->Next = t->Next;
          delete t;
     }
}
```

Xóa các khóa trong một bucket

```
void Clear_Bucket(int b)
{    Bucket t;
     while(Hash[b]->Next != NULL)
     {
          t = Hash[b]->Next;
          Hash[b]->Next = t->Next;
          delete t;
     }
}
```

Xóa tất cả khóa trong bảng băm

```
void Clear()
{
     for(int b = 0; b < M; b++)
          Clear_Bucket(b);
}
```

Duyệt các khóa trong một bucket

```
void Traverse_Bucket(int b)
{
     Bucket p = Hash[b];
     while(p->Next != NULL)
     {
          cout<<p->Next->Key<<'\t';
          p = p->Next;
     }
}
```

Duyệt tất cả các khóa trong bảng băm

```
void Traverse()
{
     for(int b = 0; b < M; b++)
     {
          cout<<"Bucket "<<b<<": ";
          Traverse_Bucket(b);
          cout<<endl;
     }
}
```

Nhập bảng băm

```
void Read_Hash()
{    int k;
     do{
          cout<<"Nhap khoa vao bang bam:";
          cin>>k;
          if(k != 0) Insert(k);
     }while (k != 0);
}
```

Hàm main( )

```
main()
{    int k, b;
     Initialize();
     Traverse();
```

```
    Read_Hash();
    Traverse();
    cout<<"Nhap khoa can tim:";
    cin>>k;
    if(Search(k))
      cout<<"Co khoa "<<k<<" trong bang bam"<<endl;
    else
      cout<<"Khong co khoa "<<k<<" trong bang bam"<<endl;
    cout<<"Nhap khoa can xoa:";
    cin>>k;
    Remove(k);
    Traverse();
    cout<<"Ban muon xoa bucket nao?";
    cin>>b;
    Clear_Bucket(b);
    Traverse();
    cout<<"Bang bam sau khi xoa:"<<endl;
    Clear();
    Traverse();
}
```

4.1.4. Các nhận xét

- Bảng băm dùng phương pháp nối kết trực tiếp sẽ "băm" N khóa vào M danh sách liên kết (bucket).
- Để tốc độ thực hiện các phép toán trên bảng băm đạt hiệu quả cao thì cần chọn hàm băm sao cho băm đều N khóa vào M bucket, lúc này trung bình mỗi bucket có N/M khóa. Chẳng hạn, phép toán search tìm kiếm tuyến tính trên bucket nên thời gian tìm kiếm lúc này có bậc là O(N/M) – nhanh hơn gấp M lần so với tìm kiếm trên một danh sách liên kết có N nút.
- Chúng ta thấy nếu chọn M càng lớn thì tốc độ thực hiện các phép toán trên bảng băm càng nhanh tuy nhiên càng chiếm nhiều bộ nhớ. Chúng ta có thể điều chỉnh M để dung hòa giữa tốc độ truy xuất và dung lượng bộ nhớ:

  + Nếu chọn M = N thì tốc độ truy xuất tương đương với truy xuất trên mảng (có bậc là O(1)), tuy nhiên tốn nhiều bộ nhớ.

  + Nếu chọn M = N/k (k = 2, 3, 4, …) thì ít tốn bộ nhớ hơn k lần, nhưng tốc độ chậm đi k lần.

4.2. Phương pháp nối kết hợp nhất

Tổ chức bảng băm theo phương pháp này là 1 danh sách đặc (mảng) mà mỗi phần tử có 2 trường:

- Trường Key: chứa khóa của phần tử
- Trường Next: chứa chỉ số của phần tử kế tiếp khi có sự đụng độ xảy ra

Khi khởi động bảng băm thì tất cả các trường key được gán NULLKEY và tất cả các trường Next được gán là -1.

0	NULLKEY	-1
1	NULLKEY	-1
2	NULLKEY	-1
3	NULLKEY	-1
4	NULLKEY	-1
5	NULLKEY	-1
	…	…
M-1	NULLKEY	-1

4.2.1 Cách biểu diễn

Khi thêm khóa k vào bảng, hàm băm HF(k) sẽ xác định địa chỉ i trong khoảng từ 0 đến M – 1.

Nếu không bị đụng độ thì thêm khóa k tại địa chỉ i này.

Nếu bị đụng độ thì địa chỉ mới được cấp phát là địa chỉ trống ở cuối bảng được quản lý bởi biến toàn cục available. Cập nhật liên kết Next sao cho các khóa bị đụng độ hình thành như một danh sách liên kết.

Khi tìm một khóa k trong bảng, hàm băm HF(k) sẽ xác định địa chỉ i trong khoảng từ 0 đến M-1, tìm khóa k trong danh sách xuất phát từ địa chỉ i.

Giả sử ta cần biểu diễn lần lượt các khóa là các số nguyên: 8, 15, 25, 30, 36, 47 vào bảng băm có tập địa chỉ (M = 11) và chọn hàm băm HF(k) = k % M.

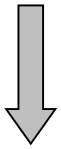
Địa chỉ tương ứng của các khóa cần biểu diễn là: 8, 4, 3, 8, 3, 3.

Ta nhận thấy có xảy ra đụng độ giữa các khóa vì chúng có cùng địa chỉ.

Giải quyết vấn đề này bằng minh họa sau đây:

0	NULLKEY	-1
1	NULLKEY	-1
2	NULLKEY	-1
3	25	-1
4	15	-1
5	NULLKEY	-1
6	NULLKEY	-1
7	NULLKEY	-1
8	8	-1
9	NULLKEY	-1
10	NULLKEY	-1

0	NULLKEY	-1
1	NULLKEY	-1
2	NULLKEY	-1
3	25	-1
4	15	-1
5	NULLKEY	-1
6	NULLKEY	-1
7	NULLKEY	-1
8	8	10
9	NULLKEY	-1
10	30	-1



0	NULLKEY	-1
1	NULLKEY	-1
2	NULLKEY	-1
3	25	9
4	15	-1
5	NULLKEY	-1
6	NULLKEY	-1
7	47	-1
8	8	10
9	36	7
10	30	-1

0	NULLKEY	-1
1	NULLKEY	-1
2	NULLKEY	-1
3	25	9
4	15	-1
5	NULLKEY	-1
6	NULLKEY	-1
7	NULLKEY	-1
8	8	10
9	36	-1
10	30	-1

4.2.2. Khai báo cấu trúc dữ liệu

```
#define NULLKEY 0
#define M 11
struct Node
{
     int Key;
     int Next;
};
Node Hash[M];
int available = M – 1;
```

4.2.3. Các phép toán

Hàm băm

Ta sử dụng hàm băm tương tự như phương pháp nối kết trực tiếp.

Khởi tạo

Khi khởi tạo bảng băm ta gán tất cả các phần tử trong bảng có trường key là NULLKEY, trường Next là -1.

```
void Initialize()
{
     for(int b = 0; b < M; b++)
     {
          Hash[b].Key = NULLKEY;
          Hash[b].Next = -1;
     }
}
```

Kiểm tra bảng băm rỗng

```
bool Empty()
{
     for(int b = 0; b < M; b++)
          if (Hash[b].Key != NULLKEY)
               return false;
     return true;
}
```

Tìm kiếm một khóa trong bảng băm

```
int Search(int k)
{    int b = HF(k);
     while(k != Hash[b].Key && Hash[b].Next != -1)
          b = Hash[b].Next;
```

```
     if(Hash[b].Key == k)
          return b;  //Tim thay
     else
          return M;  //Khong tim thay
}
```

Chọn vị trí trống cuối ở bảng băm

```
int Get_Available()
{
     while(Hash[available].Key != NULLKEY)
          available--;
     return available;
}
```

Thêm một khóa vào bảng băm

```
int Insert(int k)
{
     int b, j;
     b = Search(k);
     if(b != M)
     {
          cout<<"Khoa da co, khong them duoc"<<endl;
          return b;
     }
     b = HF(k);
     while(Hash[b].Next != -1)
          b = Hash[b].Next;
     if(Hash[b].Key == NULLKEY) //Không có đụng độ
          j = b;
     else
     {
          j = Get_Available();
          if(j < 0)
          {
               cout<<"Bang bam day, khong the them \n";
               return j;
          }
          else
               Hash[b].Next = j;
```

```
     }
     Hash[j].Key = k;
     return j;
}
```

Xóa một khóa khỏi bảng băm

```
void Remove(int k)
{
     int b = HF(k), bo = -1;
     int vt = Search(k);
     if(vt == M)
          cout<<"Khong co khoa nay"<<endl;
     else
     {
          if(vt==b)
          {
               if(Hash[vt].Next==-1)
               {
                    Hash[vt].Key = NULLKEY;
                    if(available < vt) available = vt;
               }
               else
               {
                    int t = Hash[b].Next;
                    Hash[b].Key = Hash[t].Key;
                    Hash[b].Next = Hash[t].Next;
                    Hash[t].Key = NULLKEY;
                    Hash[t].Next = -1;
                    if(available < t) available = t;
               }
          }
          else
          {
               while(Hash[b].Key != k)
               {
                    bo = b;
                    b = Hash[b].Next;
```

```
                }
                if(Hash[b].Next == -1)
                     Hash[bo].Next = -1;
                else
                {
                     Hash[bo].Next = Hash[b].Next;
                     Hash[b].Next = -1;
                }
                Hash[b].Key = NULLKEY;
                if(available < b) available = b;
           }
     }
}
```

Nhập bảng băm

```
void Read_Hash()
{
     int k;
     do{
          cout<<"Nhap khoa vao bang bam:";
          cin>>k;
          if(k != 0) Insert(k);
     }while (k != 0);
}
```

In bảng băm

```
void Print_Hash()
{
     for(int b = 0; b < M; b++){
          cout<<"Bucket "<<b<<": "<<Hash[b].Key<<'\t';
          cout<<Hash[b].Next<<endl;
     }
}
```

Hàm main( )

```
main()
{
     int k;
     Initialize();
```

```
    Print_Hash();
    Read_Hash();
    Print_Hash();
    cout<<"Nhap khoa can tim:"; cin>>k;
    int vt = Search(k);
    if(vt == M)
      cout<<"Khong co khoa "<<k<<" trong bang bam \n";
    else
      cout<<"Khoa "<<k<<" nam o vi tri "<<vt<<" trong bang \n";
    cout<<"Nhap khoa can xoa:"; cin>>k;
    Remove(k);
    Print_Hash();
}
```

4.2.4. Nhận xét

Bảng băm này chỉ tối ưu khi băm đều, mỗi danh sách chứa một vài phần tử bị đụng độ, tốc độ truy xuất lúc này có bậc O(1). Trường hợp xấu nhất là băm không đều hình thành một danh sách có N phần tử nên tốc độ truy xuất lúc này có bậc là O(N).

5. GIẢI QUYẾT SỰ ĐỤNG ĐỘ BẰNG PHƯƠNG PHÁP ĐỊA CHỈ MỞ

Phương pháp nối kết có nhược điểm là phải duy trì các danh sách và mỗi phần tử có thêm vùng liên kết. Một cách giải quyết khác là khi có sự đụng độ xảy ra thì sẽ tìm đến vị trí kế tiếp nào đó trong bảng cho đến khi tìm được vị trí trống, do đó phương pháp này được gọi là phương pháp địa chỉ mở. Dãy các chỉ số ở các bước tiếp theo phải luôn luôn như nhau đối với tất cả các khóa trong bảng.

Gọi M là số phần tử của bảng và N là số phần tử đã sử dụng. Bảng băm được gọi là đầy khi N = M – 1.

5.1. Phương pháp thử tuyến tính

5.1.1. Định nghĩa

Là phương pháp đơn giản nhất, khi có đụng độ xảy ra thì ta tìm đến vị trí kế tiếp (chỉ số tăng lên 1). Như vậy phương pháp này có G (i) = i, với G (i) là hàm tạo ra chỉ số của lần thử thứ i.

Ví dụ: Xét dãy sau đây, các khóa lần lượt được đưa vào bảng băm (M = 13).

Khóa	:	V	E	R	Y	L	O	N	G	K	E	Y
Hash	:	1	0	6	1	11	5	6	1	8	9	5

⇨ Các phần tử trong bảng băm:

0	•	E	E	E	E	E	E	E	E	E	E
1	V	V	V	V	V	V	V	V	V	V	V
2	•	•	•	Y	Y	Y	Y	Y	Y	Y	Y
3	•	•	•	•	•	•	•	G	G	G	G
4	•	•	•	•	•	•	•	•	•	•	•
5	•	•	•	•	•	O	O	O	O	O	O
6	•	•	R	R	R	R	R	R	R	R	R
7	•	•	•	•	•	•	N	N	N	N	N
8	•	•	•	•	•	•	•	•	K	K	K
9	•	•	•	•	•	•	•	•	•	E	E
10	•	•	•	•	•	•	•	•	•	•	Y
11	•	•	•	•	L	L	L	L	L	L	L
12	•	•	•	•	•	•	•	•	•	•	•

Trong đó dấu • là những vị trí còn trống trong bảng (NULLKEY)

⇨ Các chỉ số x_i dùng để thử là:

$x_0 = HF(k)$

$x_i = (x_0 + i) \% M$, với $i = 1 \rightarrow M - 1$

5.1.2 Khai báo cấu trúc dữ liệu

```
# define NULLKEY 0
# define M 13
struct Node
{
      int Key;
};
Node Hash[M];
int N = 0; //Biến toàn cục chỉ số phần tử hiện có trong bảng
```

5.1.3. Các phép toán

Hàm băm

Giả sử ta sử dụng hàm băm dạng %: HF(k) = k % M.

```
int HF(int k)
{
     return (k % M);
}
```

✧ Ta cũng có thể dùng một hàm băm bất kỳ thay cho hàm băm dạng % trên.

Khởi tạo

Khi khởi tạo bảng băm ta gán tất cả các phần tử trong bảng có trường key là NULLKEY.

```
void Initialize()
{
     for(int b = 0; b < M; b++)
          Hash[b].Key = NULLKEY;
}
```

Kiểm tra bảng băm rỗng

```
bool Empty()
{
     return (N == 0 ? true : false);
}
```

Kiểm tra bảng băm đầy

```
bool Full()
{
     return(N == M - 1);
}
```

Hàm tạo ra chỉ số để tìm vị trí trống

```
int G(int i)
{
     return i;
}
```

Tìm kiếm một khóa trong bảng băm

Tìm kiếm khóa k trên một khối đặc bắt đầu từ địa chỉ i = HF(k), nếu tìm thấy hàm trả về địa chỉ tìm thấy đó, ngược lại hàm trả về giá trị M.

```
int Search(int k)
{    if(Empty()) return M;
     int b = HF(k), vt = b, i = 0;
     while(Hash[vt].Key != NULLKEY && Hash[vt].Key != k)
     {                                   //Xử lý đụng độ
          i = i + 1;
          vt = (b + G(i)) % M;
     }
     if(Hash[vt].Key == k) return vt;
     else return M;
}
```

Thêm một khóa vào bảng băm

```
int Insert(int k)
{    int b = HF(k);
     int vt = b, i = 0;
     if(Full())
     {
          cout<<"Bang bam day, khong them duoc"<<endl;
          return M;
     }
     if(Search(k) < M)
     {
          cout<<"Khoa da co, khong them nua"<<endl;
          return M;
     }
     while(Hash[vt].Key != NULLKEY)   //Xử lý đụng độ
     {
          i = i + 1;
          vt = (b + G(i)) % M;
     }
     Hash[vt].Key = k;
     N = N + 1;
     return vt;
}
```

Xóa một khóa khỏi bảng băm

```
void Remove(int k)
{
     int vt = Search(k);
     if(vt < M)
     {
          Hash[vt].Key = NULLKEY;
          N = N - 1;
     }
     else
          cout<<"Khong co khoa nay"<<endl;
}
```

Nhập bảng băm

```
void Read_Hash()
{    int k;
     do
     {    cout<<"Nhap khoa vao bang bam:";
          cin>>k;
          if(k != 0) Insert(k);
     }while(k != 0);
}
```

In bảng băm

```
void Print_Hash()
{
     for(int b = 0; b < M; b++)
          cout<<"Bucket "<<b<<": "<<Hash[b].Key<<endl;
}
```

Hàm main( )

```
main()
{
     int k;
     Initialize();
     Print_Hash();
     Read_Hash();
     Print_Hash();
     cout<<"Nhap khoa can tim:";
     cin>>k;
     int vt = Search(k);
     if(vt == M)
       cout<<"Khong co khoa "<<k<<" trong bang bam"<<endl;
     else
       cout<<"Khoa "<<k<<" nam o vi tri "<<vt<<endl;
     cout<<"Nhap khoa can xoa:";
     cin>>k;
     Remove(k);
     Print_Hash();
}
```

5.1.4. Nhận xét

Bảng băm này chỉ tối ưu khi băm đều, trong bảng băm có các khối đặc chứa vài phần tử và các khối chưa sử dụng xen kẽ nhau, tốc độ truy xuất lúc này có bậc O(1). Trường hợp xấu nhất là băm không đều hoặc bảng băm gần đầy, lúc này hình thành một khối đặc có N phần tử nên tốc độ truy xuất lúc có có bậc là O(N).

5.2. Phương pháp thử bậc hai

5.2.1. Định nghĩa

Trong phương pháp thử tuyến tính việc tìm một vị trí trống khi xảy ra sự đụng độ rất lâu khi bảng băm gần đầy, gọi là hiện tượng gom tụ.

Để cải tiến ta cần tìm hàm G(i) sao cho có khả năng trãi đều các khóa trên các vị trí còn lại.

Để giảm bớt sự gom tụ ta dùng phương pháp thử bậc hai bằng cách chọn $G(i) = i^2$.

⇨ Các chỉ số x_i dùng để thử là:

$x_0 = HF(k)$

$x_i = (x_0 + i^2)\ \%\ M$

Với $i > 0$

Ví dụ: Xét dãy sau đây, các khóa lần lượt được đưa vào bảng băm (M = 13).

Khóa	:	V	E	R	Y	L	O	N	G	K	E	Y
Hash	:	1	0	6	1	11	5	6	1	8	9	5

⇨ Các phần tử trong bảng băm:

0	•	E	E	E	E	E	E	E	E	E	E
1	V	V	V	V	V	V	V	V	V	V	V
2	•	•	•	Y	Y	Y	Y	Y	Y	Y	Y
3	•	•	•	•	•	•	•	•	•	•	•
4	•	•	•	•	•	•	•	•	•	•	Y
5	•	•	•	•	•	O	O	O	O	O	O
6	•	•	R	R	R	R	R	R	R	R	R
7	•	•	•	•	•	•	N	N	N	N	N
8	•	•	•	•	•	•	•	•	K	K	K
9	•	•	•	•	•	•	•	•	•	E	E
10	•	•	•	•	•	•	•	G	G	G	G
11	•	•	•	•	L	L	L	L	L	L	L
12	•	•	•	•	•	•	•	•	•	•	•

5.2.2. Khai báo cấu trúc dữ liệu

Tương tự như phương pháp thử tuyến tính.

5.2.3. Các phép toán

Hàm băm và các phép toán tương tự như phương pháp thử tuyến tính tuy nhiên *hàm tạo ra chỉ số để tìm vị trí trống* và *xử lý đụng độ* khi tìm và thêm một khóa như sau:

```
int G(int i)
{
     return i * i;
}
//Xử lý đụng độ
while(Hash[vt].Key != NULLKEY && Hash[vt].Key != k)
{
     i = i + 1;
     vt = (b + G(i)) % M;
}
```

5.2.4. Nhận xét

Bảng băm này tối ưu hơn bảng băm trong phương pháp thử tuyến tính do các phần tử được phân bố đều hơn, nếu bảng băm chưa đầy tốc độ truy xuất có bậc O(1). Trường hợp xấu nhất là bảng băm gần đầy tốc độ truy xuất chậm do phải thực hiện nhiều lần so sánh.

5.3. Phương pháp băm kép

5.3.1. Định nghĩa

Ta cũng có thể tránh được hiện tương gom tụ bằng cách dùng phương pháp băm kép. Cách thực hiện này cũng tương tự như phương pháp thử tuyến tính nhưng bây giờ ta dùng thêm hàm băm thứ hai để cho một độ tăng cố định (> 1) được dùng trong các lần thử sau.

Ta đặt hàm băm thứ hai là: y = HF2(k). Như vậy ta có các chỉ số x_i dùng để thử là:

x_0 = HF(k)

$x_i = (x_0 + i * HF2(k)) \% M$, với i > 0

Ví dụ: Xét dãy sau đây, các khóa lần lượt được đưa vào bảng băm (M = 13).

Khóa	:	V	E	R	Y	L	O	N	G	K	E	Y
Hash	:	1	0	6	1	11	5	6	1	8	9	5
Hash2	:	7	3	5	6	7	2	6	3	4	8	3

⇨ Các phần tử trong bảng băm:

0	•	E	E	E	E	E	E	E	E	E	E
1	V	V	V	V	V	V	V	V	V	V	V
2	•	•	•	•	•	•	•	•	•	•	•
3	•	•	•	•	•	•	•	•	•	•	•
4	•	•	•	•	•	•	•	G	G	G	G
5	•	•	•	•	•	O	O	O	O	O	O
6	•	•	R	R	R	R	R	R	R	R	R
7	•	•	•	Y	Y	Y	Y	Y	Y	Y	Y
8	•	•	•	•	•	•	•	•	K	K	K
9	•	•	•	•	•	•	•	•	•	E	E
10	•	•	•	•	•	•	•	•	•	•	Y
11	•	•	•	•	L	L	L	L	L	L	L
12	•	•	•	•	•	•	N	N	N	N	N

5.3.2. Khai báo cấu trúc dữ liệu

Tương tự như phương pháp thử tuyến tính.

5.3.3. Các phép toán

Hàm băm

```
// Ham bam thu nhat
int HF(int k)
{
     return(k % M);
}
// Ham bam thu hai
int HF2(int k)
{
     return(k % (M - 2));
}
```

Hàm tạo ra chỉ số để tìm vị trí trống

```
int G(int i, int k)
{
     return (i * HF2(k));
}
```

Tìm kiếm một khóa trong bảng băm

Tìm kiếm khóa k trên một khối đặc bắt đầu từ địa chỉ i = HF(k), nếu tìm thấy hàm trả về địa chỉ tìm thấy đó, ngược lại hàm trả về giá trị M.

```
int Search(int k)
{
     if(Empty()) return M;
     int b = HF(k);
     int vt = b, i = 0;
     while(Hash[vt].Key != NULLKEY && Hash[vt].Key != k)
     {
          i = i + 1;
          vt = (b + G(i,k)) % M;
     }
     if(Hash[vt].Key == k)
          return vt;
     else
          return M;
}
```

Thêm một khóa vào bảng băm

```
int Insert(int k)
{
     int b = HF(k), vt = b, i = 0;
     if(Full())
     {
          cout<<"Bang bam day, khong them duoc"<<endl;
          return M;
     }
     if(Search(k) < M)
     {
          cout<<"Khoa da co, khong them nua"<<endl;
          return M;
     }
     while(Hash[vt].Key != NULLKEY)
     {
```

```
            i = i + 1;
            vt = (b + G(i,k)) % M;
        }
        Hash[vt].Key = k;
        N = N + 1;
        return vt;
}
```

✧ Các hàm *khởi tạo, kiểm tra bảng băm rỗng, kiểm tra bảng băm đầy, xóa một khóa, nhập bảng băm, xuất bảng băm* tương tự như phương pháp thử tuyến tính.

5.3.4. Nhận xét

Bảng băm này linh hoạt hơn bảng băm trong phương pháp thử tuyến tính và phương pháp thử bậc hai do sử dụng hai hàm băm khác nhau nên khả năng phân bố đều các khóa mang tính ngẫu nhiên hơn, nếu bảng băm chưa đầy tốc độ truy xuất có bậc O(1). Trường hợp xấu nhất là bảng băm gần đầy tốc độ truy xuất chậm do phải thực hiện nhiều lần so sánh.

BÀI TẬP CHƯƠNG 6

-----ᘓ✧ᘒ-----

1. Xây dựng hàm biến đổi khóa biến đổi 1 chuỗi thành 1 số nguyên tương ứng dựa vào thuật toán Horner và bảng mã được định nghĩa như sau.

Ký tự	Số nguyên	Ký tự	Số nguyên	Ký tự	Số guyên
A	1	B	4	C	7
…					
X	70	Y	73	Z	76
a	2	b	5	c	8
…					
x	71	y	74	z	77

2. Cho các phần tử như sau:

TRUONG, DAI, HOC, SU, PHAM, KY, THUAT, VINH, LONG, KHOA, CONG, NGHE, THONG, TIN

Dựa vào hàm Hash_ASCII biểu diễn các phần tử này vào bảng băm bằng các phương pháp sau, với M 17.

a. Nối kết trực tiếp

b. Nối kết hợp nhất

c. Thử tuyến tính

d. Băm kép

3. Cài đặt các yêu cầu sau với bảng băm biểu diễn bằng phương pháp nối kết trực tiếp:

a. Cho biết có bao nhiêu phần tử trong bảng không có chứa khóa nào.

b. In ra các khóa không xảy ra đụng độ.

c. Cho biết khóa lớn nhất nằm ở phần tử thứ mấy trong bảng.

d. Cho biết phần tử nào trong bảng có khóa nhiều nhất.

4. Cài đặt các yêu cầu sau với bảng băm biểu diễn bằng phương pháp thử tuyến tính:

a. Cho biết khóa K đụng độ với bao nhiêu khóa khác.

b. In ra các khóa có xảy ra đụng độ.

c. Cho biết khóa nhỏ nhất nằm ở phần tử thứ mấy trong bảng.

d. In ra các vị trí còn trống trong bảng.

5. Cài đặt các yêu cầu sau cho bảng băm biểu diễn bằng phương pháp nối kết hợp nhất, phương pháp thử bậc hai và phương pháp băm kép với khóa là chuỗi ký tự:

a. Khai báo cấu trúc dữ liệu.

b. Xây dựng hàm băm tương ứng.

c. Thêm, tìm và xóa 1 khóa.

d. Nhập và xuất bảng.

TÀI LIỆU THAM KHẢO

-----ଓ✧ଌ-----

[1] Data Structures And Algorithm Analysis In Cpp_2014.

[2] AdamDrozdek_DataStructures_and_Algorithms_in_C_4Ed.

[3] Nguyễn Hồng Chương, *Cấu trúc dữ liệu ứng dụng và cài đặt bằng C,* Nhà xuất bản Thành phố Hồ Chí Minh.

[4] Lê Xuân Trường biên dịch, *Giáo trình Cấu trúc dữ liệu bằng ngôn ngữ C++,* Nhà xuất bản Thống Kê.

[5] Đỗ Xuân Lôi, *Cấu trúc dữ liệu và giải thuật,* Nhà xuất bản Khoa học Kỹ thuật.

[6] Robert Sedgewick, *Cẩm nang Thuật Toán Vol.1,* Nhà xuất bản Khoa học Kỹ thuật.

[7] Robert Sedgewick, *Cẩm nang Thuật Toán Vol.2,* Nhà xuất bản Khoa học Kỹ thuật.

[8] Lê Minh Trung, *Bài tập Cấu trúc dữ liệu & Thuật toán,* Nhà xuất bản thống kê.

[9] Gia Việt biên dịch, *Bài tập nâng cao cấu trúc dữ liệu cài đặt bằng C,* Nhà xuất bản Thống Kê.

[10] Phạm Văn Ất, *Giáo trình kỹ thuật lập trình C Căn bản & Nâng cao,* Nhà xuất bản Hồng Đức.

[11] Nguyễn Thanh Thủy, Nguyễn Quang Huy, *Bài tập lập trình ngôn ngữ C,* Nhà xuất bản Khoa học Kỹ thuật.

[12] Các trang Web có liên quan đến môn học.

MỤC LỤC

-----ଓ✧ଌ-----