TRƯỜNG ĐẠI HỌC SƯ PHẠM ĐÀ NẴNG

KHOA TIN HỌC

----- ☙🕮❧ -----

GIÁO TRÌNH

CẤU TRÚC DỮ LIỆU VÀ GIẢI THUẬT

GIẢNG VIÊN: PHẠM ANH PHƯƠNG

ĐÀ NẴNG 08/2018

MỤC LỤC

CHƯƠNG 1: CẤU TRÚC DỮ LIỆU VÀ GIẢI THUẬT

1.1. CẤU TRÚC DỮ LIỆU

1.1.1. Các khái niệm và định nghĩa

1.1.1.1. Khái niệm dữ liệu

Trong máy tính, dữ liệu là thông tin đã được chuyển sang một định dạng thuận tiện hơn để có thể xử lý. Đối với khoa học máy tính ngày nay và phương tiện truyền thông, dữ liệu là thông tin chuyển đổi thành dạng số nhị phân.

1.1.1.2. Khái niệm cấu trúc lưu trữ

Cách biểu diễn một cấu trúc dữ liệu trong bộ nhớ máy tính được gọi là cấu trúc lưu trữ. Có thể có nhiều cấu trúc lưu trữ khác nhau cho một cấu trúc dữ liệu. Chẳng hạn một cấu trúc dữ liệu kiểu danh sách ta có thể lưu trữ dữ liệu bằng các ô nhớ liên tiếp nhau trong bộ nhớ hoặc có thể lưu trữ bằng các ô nhớ không liên tiếp nhau trong bộ nhớ.

Có thể có nhiều cấu trúc dữ liệu khác nhau được cài đặt trong bộ nhớ bằng một cấu trúc lưu trữ. Chẳng hạn cấu trúc xâu ký tự, cấu trúc mảng đều có thể cài đặt trong bộ nhớ bằng các ô nhớ liên tiếp nhau.

1.1.2. Lựa chọn dữ liệu cho bài toán

Việc chọn cấu trúc dữ liệu thích hợp để tổ chức dữ liệu vào ra và trên cơ sở đó xác lập một giải thuật để đưa ra kết quả mong muốn là một khâu rất quan trọng.

Chọn một cấu trúc dữ liệu phải xét tới các phép toán tác động lên cấu trúc dữ liệu đó và ngược lại xét đến phép toán phải chú ý phép toán đó tác động trên cấu trúc dữ liệu nào. Bởi vì có phép toán hữu hiệu đối với cấu trúc dữ liệu này nhưng không hữu hiệu với cấu trúc dữ liệu kia.

1.2. GIẢI THUẬT

1.2.1. Khái niệm, các tính chất của giải thuật

1.2.1.1. Khái niệm giải thuật

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

Giải thuật (algorithrm) là một dãy các câu lệnh (statement) chặt chẽ và rõ ràng xác định một trình tự các thao tác trên một số đối tượng nào đó sao cho sau một số hữu hạn bước thực hiện ta đạt được kết quả mong muốn.

Trong cuộc sống hàng ngày chúng ta gặp rất nhiều thuật giải cho một bài toán.

1.2.1.2. Các tính chất của giải thuật

Tính xác định

Tính xác định đòi hỏi, ở mỗi bước của thuật toán, các thao tác đều phải **rõ ràng, không thể gây ra sự nhập nhằng, lẫn lộn**. Nói khác đi là trong cùng một điều kiện, hai bộ xử lý (người hoặc máy) thực hiện cùng một bước của thuật toán thì phải cho cùng một kết quả. Hơn thế nữa, các bộ xử lý thuật toán không cần phải hiểu được ý nghĩa của các bước thao tác này.

Tính kết thúc (tính dừng)

Thuật toán bao giờ cũng phải dừng sau một số hữu hạn bước thực hiện.

Tính đúng đắn

Yêu cầu bắt buộc nữa của thuật toán là tính đúng đắn, hiểu theo nghĩa là: với dữ liệu vào cho trước, thuật toán thực hiện sau một số hữu hạn bước cho trước sẽ dừng và cho kết quả đúng của bài toán. Kết quả mong muốn thường được xác định qua định nghĩa.

Tính phổ dụng

Thuật toán thường được xây dựng không chỉ để giải một bài toán riêng lẻ mà là một lớp các bài toán có cùng cấu trúc với những dữ liệu cụ thể khác nhau và luôn luôn dẫn đến kết quả mong muốn.

Tính hiệu quả

Tính hiệu quả được đánh giá dựa trên một số tiêu chuẩn nhất định như **khối lượng tính toán, thời gian và không gian** được sử dụng bởi thuật toán (khi giải bằng máy),...

Các đại lượng vào (input) và ra (output)

Một thuật toán có thể có nhiều đại lượng vào mà ta thường gọi là dữ liệu vào.

Sau khi dừng thuật toán thì tùy theo chức năng của thuật toán mà ta có thể thu được một số đại lượng ra xác định. Các đại lượng ra cũng thường được gọi là dữ liệu ra hay kết quả.

1.2.2. Mối quan hệ của cấu trúc dữ liệu và giải thuật

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

Theo cách tiếp cận của lập trình có cấu trúc, NiklausWirth đưa ra công thức thể hiện được mối liên hệ giữa cấu trúc dữ liệu và giải thuật như sau:

THUẬT TOÁN + CẤU TRÚC DỮ LIỆU = CHƯƠNG TRÌNH

(Algorithms + Data Structures = Programs)

Khi cấu trúc dữ liệu của bài toán thay đổi thì giải thuật cũng phải thay đổi theo cho phù hợp với cách thức tổ chức dữ liệu mới. Ngược lại trong quá trình xây dựng việc hoàn thiện giải thuật cũng gợi mở cho người lập trình cách tổ chức dữ liệu phù hợp với giải thuật và tiết kiệm tài nguyên hệ thống.

Quá trình giải một bài toán trên máy tính chúng ta phải chú ý đến mối liên hệ mật thiết giữa giải thuật và cấu trúc dữ liệu. Vì thế khi tiến hành nghiên cứu về cấu trúc dữ liệu cho bài toán thì đồng thời phải xác lập các giải thuật ứng dụng cho cấu trúc dữ liệu đó.

Trong cuộc sống hàng ngày chúng ta gặp rất nhiều trường hợp cho thấy rõ mối liên hệ giữa cấu trúc dữ liệu và giải thuật.

1.3. ĐỘ PHỨC TẠP CỦA THUẬT TOÁN

1.3.1. Khái niệm

- Thước đo hiệu quả của một thuật toán là ***thời gian*** để giải bài toán và ***không gian*** nhớ đòi hỏi để thực hiện thuật toán.
- Độ phức tạp thời gian của thuật toán có thể được đánh giá thông qua số ***phép toán tích cực (phép toán được dùng không ít hơn các phép toán khác)*** khi các giá trị đầu vào có kích thước xác định.

1.3.2. Tiêu chuẩn đánh giá

Định nghĩa 1: Hàm f(n) có cấp bé hơn hoặc bằng hàm g(n) nếu $\exists C>0$ và số tự nhiên n_0 sao cho:

$$|f(n)| \le C|g(n)| \text{ với mọi } n \ge n_0.$$

Ký hiệu: f(n)=O(g(n)) và gọi f(n) thoả mãn ***quan hệ big-O*** đối với g(n).

Ví dụ 1: Hàm $f(n)=\frac{n(n+3)}{2}$ là hàm bậc hai và hàm bậc hai đơn giản nhất là n^2. Ta có:
$f(n)= \frac{n(n+3)}{2} = O(n^2)$ vì $\frac{n(n+3)}{2} \le n^2 \ \forall n \ge 3$ (C=1, n_0=3).

Định nghĩa 2: Nếu thuật toán có độ phức tạp là f(n) với f(n)=O(g(n)) thì ta nói thuật toán có độ phức tạp **O(g(n))**.

Mệnh đề: Cho $f_1(n)=O(g_1(n))$ và $f_2(n)=O(g_2(n))$. Khi đó:

- $(f_1 + f_2)(n) = O(\max(|g_1(n)|,|g_2(n)|)$ (Qui tắc tổng)
- $(f_1 f_2)(n) = O(g_1(n)g_2(n))$ (Qui tắc nhân)

Ví dụ 2**:** Cho đoạn chương trình sau (k<m<n).

```
int d=0;
for (int i=1;i<=k;i++) d++;
for (int j=1;j<=m;j++) d++;
for (int h=1;h<=n;h++) d++;
```

Với các vòng lặp rời nhau, theo quy tắc tổng, độ phức tạp của đoạn chương trình trên sẽ là: O(max(k,m,n)) = O(n)

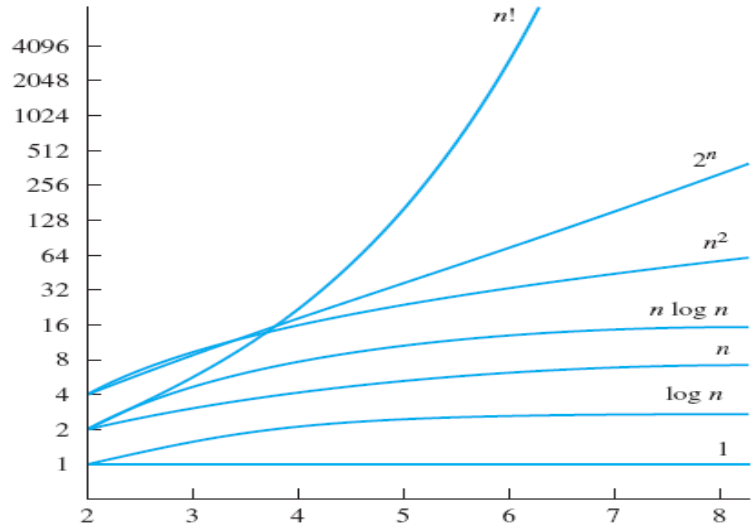
Ví dụ 3: Cho đoạn chương trình sau

```
int d=0;
for (int i=1;i<=n;i++)
  for (int j=1;j<=n;j++)
     for (int h=1;h<=n;h++) d++;
```

Với các vòng lặp lồng nhau, theo quy tắc nhân, độ phức tạp của đoạn chương trình trên sẽ là: O(n×n×n) = O(n^3)

1.3.3. Độ phức tạp của một số thuật toán thông dụng

- Thuật toán tìm kiếm tuyến tính: O(n)
- Thuật toán tìm kiếm nhị phân: O($\log_2 n$)
- Thuật toán sắp xếp chọn, chèn, nổi bọt: O(n^2)
- Thuật toán Quick sort: O($n\log_2 n$)
- Liệt kê dãy nhị phân độ dài n: O(2^n)
- Liệt kê các hoán vị của n phần tử: O(n!)



1.4. GIẢI THUẬT ĐỆ QUY

1.4.1. Khái niệm

Một đối tượng được gọi là đệ quy nếu nó hoặc một phần của nó được định nghĩa thông qua khái niệm về chính nó.

Một chương trình gọi là đệ quy nếu trong chương trình có lời gọi đến chính nó.

Tổng quát:

Nếu một lời giải của bài toán T được thực hiện bằng lời giải của bài toán T', có dạng giống như T thì đó là một lời giải đệ quy. Giải thuật tương ứng với lời giải như vậy gọi là

giải thuật đệ quy. Nếu giải thuật ấy được viết dưới dạng một thủ tục thì thủ tục ấy được gọi là thủ tục đệ quy.

1.4.2. Cấu trúc của giải thuật đệ quy

Một giải thuật đệ quy bao giờ cũng gồm có hai thành phần:

Thành phần dừng (phần neo): Không chứa khái niệm đang định nghĩa. Phần này xác định điểm dừng của giải thuật đệ quy. Trường hợp này còn được gọi là trường hợp suy biến. Nếu một giải thuật đệ quy không có trường hợp suy biến thì sẽ dẫn đến lặp vô hạn và sinh lỗi khi thực thi chương trình.

Thành phần đệ quy: Có chứa khái niệm đang định nghĩa. Trường hợp này là phân tích và xây dựng trường hợp chung của bài toán (nghĩa là đưa bài toán về cùng loại nhưng với dữ liệu có kích thước "nhỏ" hơn và mục đích là đưa bài toán tiến dần về phần neo).

1.4.3. Phương pháp xây dựng giải thuật đệ quy

Khi xây dựng giải thuật đệ quy ta tiến hành những bước sau:

Bước 1: Tham số hóa bài toán.

Bước 2: Xác định trường hợp suy biến.

Bước 3: Phân tích và xây dựng trường hợp chung của bài toán (thành phần đệ quy) có nghĩa là đưa bài toán về dạng bài toán cùng loại nhưng với kích thước dữ liệu nhỏ hơn.

Ví dụ 1: Viết hàm đệ qui để tính n! = 1.2...n.

- Tham số hóa: n! = Factorial(n);
- Factorial(0) = 1 (trường hợp suy biến)
- Factorial(n) = n*Factorial(n-1) (trường hợp chung)

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

Ví dụ 2: Viết hàm in ra biểu diễn nhị phân của một số nguyên.

- Tham số hóa: Bin(n);
- n=0: kết thúc (trường hợp suy biến)
- n>0: lưu lại **cout<<n%2;** và thực hiện tiếp **Bin(n/2)** (trường hợp chung)

```
void Bin(int n)
{
     if (n> 0)
     {
          Bin(n/2)
```

```
            cout<<n%2;
        }
    }
```

Ví dụ 3: Bài toán tháp Hà Nội **(Lucas' Tower** 1883)

Có n đĩa với kích thước nhỏ dần xếp chồng lên nhau. Đĩa to dưới, đĩa nhỏ trên. Yêu cầu bài toán: Chuyển chồng đĩa từ cọc A sang cọc C với điều kiện sau:

- Mỗi lần chỉ được chuyển một đĩa.
- Được phép dùng cọc B làm cọc trung gian để trung chuyển.
- Không được đặt đĩa to nằm ở trên đĩa nhỏ.

Bước 1: Tham số hóa bài toán.

Gọi hàm ***ThapHN(n, A, B, C)*** là hàm chuyển n đĩa từ cọc A sang cọc C (lấy cọc B làm trung gian).

Bước 2: Trường hợp suy biến

Với n = 1: Chuyển đĩa từ cột A sang cột C là xong.

Bước 3: Phân tích trường hợp tổng quát.

* **Xét trường hợp n=2**: thực hiện 3 phép chuyển:

- Chuyển đĩa thứ nhất từ cọc A sang cọc B: ThapHN(1,A,C,B);
- Chuyển đĩa thứ hai từ cọc A sang cọc C: ThapHN(1,A,B,C);
- Chuyển đĩa thứ nhất từ cọc B sang cọc C: ThapHN(1,B,A,C);

* **Tổng quát hóa với n≥2**: Xem (n-1) đĩa nằm ở trên đóng vai trò như 1 đĩa thì có thể hình dung đang có 2 đĩa trên cọc A. Nếu mô phỏng như trường hợp n = 2, giải thuật chuyển như sau:

- Chuyển (n-1) đĩa từ cọc A sang cọc B: ThapHN(n-1,A,C,B);
- Chuyển 1 đĩa từ cọc A sang cọc C: ThapHN(1,A,B,C);
- Chuyển (n-1) đĩa từ cọc B sang cọc C: ThapHN(n-1,B,A,C);

Như vậy, bài toán tháp Hà Nội có thể cài đặt như sau:

```
void ThapHN(int n,char A,char B,char C)
{
  if(n==1) cout<<endl<<"Chuyen dia tu "<<A<<" qua "<<C;
  else
       {
         ThapHN(n-1,A,C,B);
         ThapHN(1,A,B,C);
         ThapHN(n-1,B,A,C);
       }
}
```

1.4.4. Ưu và nhược điểm của đệ quy

Ưu điểm

- Cung cấp cho ta cơ chế giải quyết bài toán phức tạp một cách đơn giản.
- Chương trình trong sáng, ngắn gọn.
- Dễ dàng chuyển thành mã lệnh trên các ngôn ngữ lập trình.

Nhược điểm

- Tốn bộ nhớ
- Mất nhiều thời gian xử lý nên giảm tốc độ chạy chương trình.
- Không thể áp dụng cho mọi ngôn ngữ ví dụ như Fortran.

1.5. GIẢI THUẬT QUAY LUI

*** Bài toán**: Xây dựng các bộ giá trị gồm n phần tử ($x_1,...,x_n$) từ một tập hữu hạn cho trước sao cho các bộ đó thỏa mãn một yêu cầu B nào đó.

*** Phương pháp**:

Giả sử đã xác định được k-1 phần tử đầu tiên của bộ giá trị là $x_1,...,x_{k-1}$. Cần xác định phần tử thứ k tiếp theo, phần tử này được xác định theo cách sau:

- Giả sử T_k là tập tất cả các giá trị mà x_k có thể nhận. Vì T_k hữu hạn nên có thể đặt n_k là số phần tử của T_k theo một thứ tự nào đó, nghĩa là có thể thành lập một ánh xạ 1-1 từ tập T_k lên tập $\{1,2,...,n_k\}$
- Xét $j \in \{1,2,...,n_k\}$, ta nói rằng "**j chấp nhận được**" nếu có thể bổ sung phần tử thứ j trong T_k với tư cách là phần tử x_k vào dãy $\mathbf{x_1,...,x_{k-1}}$ để được dãy $\mathbf{x_1,...,x_k}$.
- Nếu k = n: bộ ($x_1,...,x_k$) thỏa nãm yêu cầu B $\Rightarrow$ bộ này sẽ được liệt kê.
- Nếu k < n: cần lặp lại quá trình trên, tức là phải bổ sung tiếp phần tử x_{k+1} vào dãy $x_1,...,x_k$.

✍ NHẬN XÉT:

Nét đặc trưng của giải thuật quay lui là muốn có được lời giải thì phải đi từng bước bằng phép thử. Khi một bước lựa chọn thỏa mãn thì cần ghi nhận kết quả và tiến hành các bước tiếp theo. Ngược lại, khi không có một lựa chọn nào thỏa mãn thì phải ***quay lui bước trước đó***, xóa bớt các trường hợp mà ta đã đi qua và thử lại với các lựa chọn còn lại.

Sau đây là hàm đệ qui mô tả giải thuật quay lui:

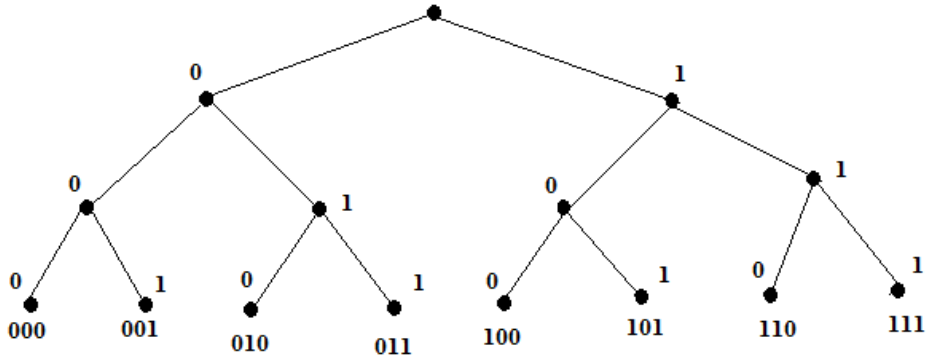
```
void  Thu(int k)
{
     int j;
     for(j=1;j<=nk;j++) //Duyệt qua các giá trị của Tk
     {
          <Xác định xk theo j>;
          if(k==n) <Ghi nhận 1 bộ giá trị>;
```

```
            else Thu(k+1); //Quay lui
        }
    }
```

Ví dụ 1: Viết chương trình liệt kê các dãy nhị phân có độ dài n.

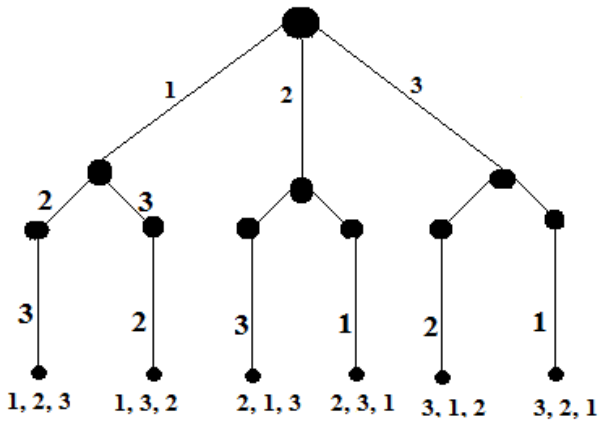
```
#include <iostream.h>
#include <conio.h>
int n,x[20];
void InKetQua()
{
  for(int i=1;i<=n;i++) cout<<x[i];
  cout<<"\n";
}
void  Thu(int k)  //Tìm phần tử x_k
{
    for(int j=0;j<=1;j++)
    {
        x[k]=j;        //Xác định phần tử x_k theo j
        if(k==n) InKetQua();
        else Thu(k+1); //Quay lui: tìm phần tử x_{k+1}
    }
}
int main()
{
 cout<<"\n Nhap do dai xau nhi phan: "; cin>>n;
 Thu(1);
 getch();
 return 0;
}
```

Với n = 3 thì kết quả nhận được như sau:



Vi dụ 2: Viết chương trình liệt kê các hoán vị của {1,2,...,n}.

- Biểu diễn các hoán vị dưới dạng $x_1,x_2,...,x_n$, trong đó $x_i \in [1,n]$ và $x_i \neq x_j$ $(i \neq j)$.
- Các giá trị từ 1 tới n sẽ lần lược đề cử cho p_i, trong đó $j \in [1,n]$ được chấp nhận ***nếu nó chưa được dùng đến***. Vì vậy cần tạo ra một dãy biến logic b_j để xét xem j đã được dùng hay chưa.
- Gán b_j = TRUE nếu j chưa được dùng, ngược lại b_j = FALSE.
- Ta có thể hình dung bài toán như hình vẽ sau: Với n=3 thì bài toán trở thành liệt kê các hoán vị của các phần tử 1, 2, 3. Các hoán vị được liệt kê theo thứ tự từ điển tăng dần như hình vẽ sau:




```
#include <iostream.h>
#include <conio.h>
int n,b[20],x[20];
void Init()
{
 cout<<"\n Nhap so nguyen duong n = "; cin>>n;
 for(int i=1;i<=n;i++) b[i]=1; //Chưa dùng
}
void InKetQua()
{
```

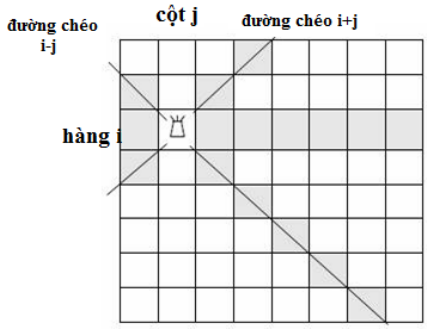
```
  for(int i=1;i<=n;i++) cout<<x[i];
  cout<<"\n";
}
void  Thu(int k) //Tìm phần tử x_k
{
     int j;
     for(j=1;j<=n;j++)
     {
          if(b[j]) // phần tử j chưa sử dụng
          {
               x[k]=j; //Xác định phần tử x_k theo j
               b[j]=0; //phần tử j đã sử dụng
               if(k==n) InKetQua();
               else Thu(k+1); //Quay lui: Tìm phần tử x_{k+1}
               b[j]=1; //Trả lại trạng thái cũ (j chưa sử dụng)
          }
     }
}
int main()
{
 Init();
 Thu(1);
 getch();
}
```

Ví dụ 3: (**Bài toán xếp hậu)** Liệt kê các cách xếp n quân Hậu trên bàn cờ n x n sao cho chúng không ăn được lẫn nhau.

Cách giải:

Đánh số cột và dòng của bàn cờ là từ 1 cho đến n. Mỗi dòng chỉ xếp 1 quân hậu. Như vậy vấn đề của chúng ta bây giờ là tìm xem mỗi quân hậu trên mỗi dòng sẽ được xếp vào cột nào trên bàn cờ thì thỏa mãn là chúng không ăn được lẫn nhau.

Theo như luật chơi cờ thì quân Hậu ăn theo đường ngang, dọc và hai đường chéo. Như vậy yêu cầu bài toán được đưa về dạng tìm các cách xếp hậu trên bàn cờ hay nói cách khác là tìm các bộ gồm n thành phần (x_1, x_2,…,x_n) trong đó xi = j nghĩa là quân hậu của hàng i được xếp vào cột j. Như vậy các giá trị đề cử cho xi là từ 1 cho đến n. Giá trị j này được chấp nhận nếu các quân hậu trước đó chưa chiếu đến (hay nói cách khác là vị trí ô (i,j) này chỉ được chấp nhận khi các quân hậu đã xếp trước đó không ăn được).



Các nước chiếu của quân hậu

Xét n=8

Đối với mỗi ô trong hàng, sẽ có 1 cột và 2 đường chéo đi qua nó là đường chéo trái và đường chéo phải.

Ta sẽ dùng 3 mảng kiểu boolean để biểu thị cho các cột, các đường chéo trái, và các đường chéo phải (có tất cả 15 đường chéo trái và 15 đường chéo phải).

int a[8]; với các giá trị ban đầu của nó là true hay a[i]=1

int b[15], c[15];với các giá trị ban đầu của nó là true hay b[i]=1, c[i]=1

Trong đó:

a[i] = true: hàng i chưa bị chiếm bởi quân hậu nào.

b[k]=true: Đường chéo trái k chưa bị chiếm bởi quân hậu nào với 2<=k<=2*n

c[k] = true: Đường chéo phải k chưa bị chiếm bởi quân hậu nào với 1-n<=k<=n-1

Chú ý rằng các ô (i, j) cùng nằm trên 1 đường chéo trái thì có cùng giá trị i + j, và cùng nằm trên đường chéo phải thì có cùng giá trị i - j.

Để kiểm tra xem ô (i, j) có an toàn không, ta chỉ cần kiểm tra xem hàng i và các đường chéo (i +j), (i - j) đã bị chiếm chưa, tức là kiểm tra a[i], b[i + j], và c[i - j].

Ngoài ra, ta cần có 1 mảng x để lưu giữ chỉ số hàng của quân hậu trong cột j.

int x[8];

Vậy với thao tác đặt hậu vào vị trí hàng i cột j, ta cần thực hiện các công việc:

x[j] = i; a[i] = false; b[i + j] = false; c[i - j] = false;

Với thao tác bỏ hậu ra khỏi cột j trong hàng i, ta cần thực hiện các công việc:

a[i] = true; b[i + j] = true; c[i - j] = true;

Còn điều kiện để kiểm tra xem vị trí tại cột j trong hàng i có an toàn không là:

(a[i] = =true) && (b[i + j] == true) && (c[i - j] == true)

Như vậy, hàm Try( ) sẽ được thể hiện cụ thể bằng ngôn ngữ C như sau:

```
void Try(int j)
{
  for(int i=1;i<=n;i++)
    if(a[i]&&b[i+j]&&c[i-j])
    {
     x[j]=i;
```

```
      a[i]=false;
      b[i+j]=false;
      c[i-j]=false;
      if(j==8)In_kq();
      else Try(j+1);
      a[i]=true;
      b[i+j]=true;
      c[i-j]=true;
     }
}
void In_kq()
{
  cout<<"\n -Cach xep: "<<++d<<":\n";
  for(int i=1;i<=n;i++)
     cout<<"("<<i<<","<<x[i]<<") ";
  cout<<"\n";
}
```

BÀI TẬP

Bài 1: Viết hàm tìm ước chung lớn nhất của hai số nguyên dương a, b.

Bài 2: Viết hàm đếm số chữ số của một số nguyên dương n theo hai cách: đệ quy và không đệ qui.

Bài 3: Dãy số Fibonacci được định nghĩa như sau:

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

Viết hàm để tính F(n) theo hai cách: đệ quy và không đệ qui.

Bài 4: Xét định nghĩa đệ qui

$$A(m,n) = \begin{cases} n+1 & , \quad m=0 \\ A(m-1,1) & , \quad n=0 \\ A(m-1, A(m,n-1)) & , \quad m>0 \wedge n>0 \end{cases}$$

Hàm này được gọi là hàm Ackermann.

1. Tính A(1,2)?

2. Viết hàm đệ qui để tính A(m,n).

Bài 5: Một số nguyên dương được gọi là đối xứng nếu chữ số thứ nhất bằng chữ số cuối, chữ số thứ hai bằng chữ số gần cuối,... Viết hàm đệ qui để kiểm tra số nguyên dương n có phải là số đối xứng hay không.

Bài 6: Viết các hàm đệ qui và không đệ qui để tính:

$S_1 = 1+2+3+.....+n$;

$S_2 = 1+1/2 + ...+ 1/n$;

$S_3 = 1-1/2 +.....+ (-1)^{n+1}\ 1/n$

$S_4 = 1 + \sin(x) + \sin^2(x) + ...+ \sin^n(x)$

Bài 7: Viết hàm đệ quy để tính C^k_n biết :

$C^n_n = 1$, $C^0_n = 1$, $C^k_n = C^{k-1}_{n-1} + C^k_{n-1}$.

Bài 8: Viết hàm để in ra màn hình số đảo ngược của một số nguyên cho trước theo hai cách: đệ qui và không đệ qui.

Bài 9*: Viết chương trình cài đặt bài toán 8 quân hậu.

CHƯƠNG 2: DANH SÁCH ĐẶC

2.1. DANH SÁCH

2.1.1. Khái niệm danh sách

Danh sách là một tập hợp gồm nhiều phần tử (element) $a_1a_2…a_n$ mà tính chất cấu trúc của nó là mối liên hệ tương đối giữa các phần tử với nhau: nếu biết được phần tử ai thì ta sẽ biết được vị trí của các phần tử a_{i+1}.

Số phần tử của danh sách được gọi là chiều dài của danh sách. Một danh sách có chiều dài bằng 0 là danh sách rỗng.

Một tính chất quan trọng của danh sách là các phần tử có thể được sắp xếp tuyến tính theo vị trí của chúng trong danh sách.

2.1.2. Danh sách đặc

Danh sách đặc (condensed list) là một danh sách mà các phần tử được sắp xếp kế tiếp nhau trong bộ nhớ, đứng ngay sau vị trí phần tử a_i là vị trí phần tử a_{i+1}.

2.2. CÁC THAO TÁC TRÊN DANH SÁCH ĐẶC

Thông thường, ta sử dụng mảng một chiều để lưu trữ danh sách đặc. Giả sử danh sách được lưu trong mảng A[ ] có n phần tử.

2.2.1. Duyệt danh sách

Duyệt danh sách là công việc mà chúng ta cần phải thăm tất cả các phần tử của danh sách mà không được bỏ sót phần tử nào. Thông thường ta dùng vòng lặp để duyệt qua tất cả các phần tử của danh sách.

Giải thuật duyệt mảng có thể được thực hiện như sau:

```
for (int i=1; i<=n; i++) <Thăm A[i]>;
```

2.2.2. Chèn một phần tử vào danh sách

- Đầu vào: Mảng A có n phần tử, vị trí chèn k, giá trị cần chèn X.
- Đầu ra: Mảng A có n+1 phần tử.

Giải thuật:

Bước 1: Dời các phần tử của mảng A từ vị trí thứ k sang phải một vị trí

```
for(i=n; i>=k; i--) A[i+1] = A[i];
```

Bước 2: Chèn giá trị X vào vị trí k

```
A[k] = X;
```

Bước 3: Tăng kích thước của mảng lên 1 đơn vị

```
n++;
```

2.2.3. Xóa 1 phần tử ra khỏi mảng

Muốn xóa một phần tử tại vị trí k trong mảng A có n phần tử (1<=k<=n), ta thực hiện giải thuật như sau:

Bước 1: Dời các phần tử của mảng A từ vị trí thứ k+1 qua trái một vị trí.

for (i=k; i<n; i++) A[i] =A[i+1];

Bước 2: Số phần tử của mảng sẽ giảm bớt 1.

n—

2.3. ƯU VÀ NHƯỢC ĐIỂM KHI DÙNG MẢNG

2.3.1. Ưu điểm

- Tốc độ truy cập nhanh.
- Tổ chức và cài đặt đơn giản.

2.3.2. Nhược điểm

- Số phần tử của mảng phải được xác định trước nên gây lãng phí không gian lưu trữ nếu ta không sử dụng hết số lượng đã khai báo.
- Nếu việc chèn và xóa các phần tử diễn ra liên tục thì tốc độ xử lý sẽ rất chậm.

2.4. MẢNG 2 CHIỀU

2.4.1. Khai báo mảng 2 chiều

<Kiểu> <Tên_biến_mảng> [Số_dòng][số_cột];

Ví dụ:

```
float a[3][4];
int b[10][10];
```

2.4.2. Truy cập mảng 2 chiều

Để truy cập đến phần tử [i,j] (dòng i cột j) trong mảng hai chiều M, ta sử dụng cú pháp: M[i][j].

Ví dụ: Nhập vào mảng 2 chiều m dòng, n cột. Tính và in ra màn hình tổng các phần tử lớn nhất của mỗi dòng.

```
#include <iostream.h>
#define MaxM 10
#define MaxN 10
void Nhap(int &m,int &n,int a[MaxM][MaxN])
{
  cout<<"So dong: "; cin>>m;
```

```
  cout<<"So cot: "; cin>>n;
  for(int i=0;i<m;i++)     //dong
    for(int j=0;j<n;j++)   //cot
     {
       cout<<"a["<<i<<","<<j<<"]=";
       cin>>a[i][j];
     }
}
void InMang(int m,int n,int a[MaxM][MaxN])
{
  for(int i=0;i<m;i++)      //số dòng
  {
   for(int j=0;j<n;j++)          //số cột
       printf("%4d",a[i][j]);
   cout<<"\n";
  }
     getch();
}
int Tong(int m,int n,int a[MaxM][MaxN])
{
  int s=0;
  for(int i=0;i<m;i++) //duyệt qua từng dòng
  {
     //tìm phần tử lớn nhất của dòng i
    int max=a[i][0];
    for(int j=1;j<n;j++)    //duyệt qua n phần tử của dòng i
      if(max<a[i][j]) max=a[i][j];
         //cong vao bien s
    s+=max;
  }
  return s;
}
main()
```

```
{
  int m,n,a[MaxM][MaxN];
  Nhap(m,n,a);
  InMang(m,n,a);
  cout<<"Tong max: "<<Tong(m,n,a);
  getch();
}
```

2.5. SẮP XẾP VÀ TÌM KIẾM

2.5.1. Một số thuật toán sắp xếp

2.5.1.1. Sắp xếp nổi bọt (Bubble sort)

Ý tưởng:

Đi từ cuối mảng về đầu mảng, trong quá trình đi nếu phần tử ở dưới (phía sau) nhỏ hơn phần tử đứng ngay trên (trước) nó thì theo nguyên tắc của bọt khí "phần tử nhẹ" sẽ "trồi" lên phía trên "phần tử nặng" (hai phần tử này sẽ được đổi chỗ cho nhau). Kết quả là phần tử nhỏ nhất (nhẹ nhất) sẽ được đưa lên (trồi lên) trên bề mặt (đầu mảng).

Sau mỗi lần đi chúng ta sẽ được một phần tử trồi lên đúng chỗ. Như vậy sau n-1 lần đi thì tất cả các phần tử trong mảng được sắp xếp theo thứ tự tăng dần.

Cài đặt thuật toán:

```
void BubbleSort(int a[], int n)
{
     for (int i=0; i<n-1; i++)
          for (int j=n-1; j>i; j--)
              if (a[j]<a[j-1])
                   Swap(a[j],a[j-1]);
}
```

2.5.1.2. Sắp xếp chọn (Selection sort)

Ý tưởng:

Giả sử dãy a có n phần tử chưa có thứ tự. Chọn phần tử có giá trị nhỏ nhất trong n phần tử chưa có thứ tự này để đưa lên đầu nhóm có n phần tử.

Tiếp tục chọn phần tử có giá trị nhỏ nhất trong n -1 phần tử chưa có thứ tự này để đưa lên đầu nhóm của n-1 phần tử,…

Sau n-1 lần lựa chọn phần tử nhỏ nhất để đưa lên đầu nhóm thì tất cả các phần tử trong dãy a sẽ có thứ tự tăng dần.

Cài đặt thuật toán:

```
void Selection_Sort(int a[], int n)
{
```

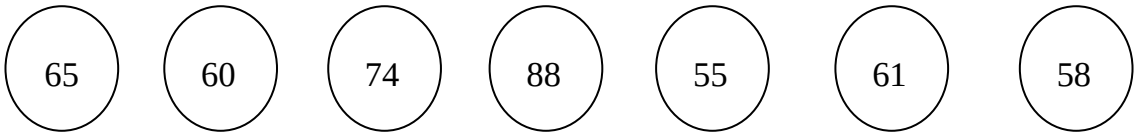
```
    for (int i=0; i<n-1; i++)
    {
        // tim phan tu min tu n phan tu
        int min = a[i], min_pos = i;
        for(int j=i+1; j<n; j++)
        if (a[j]<min)
        {
            min =a[j];
            min_pos = j;
        }
        //Swap(a[i],a[min_pos]);
        int temp = a[min_pos];
        a[min_pos]= a[i];
        a[i]=temp;
    }
}
```

2.5.1.3. Sắp xếp nhanh (Quick sort)

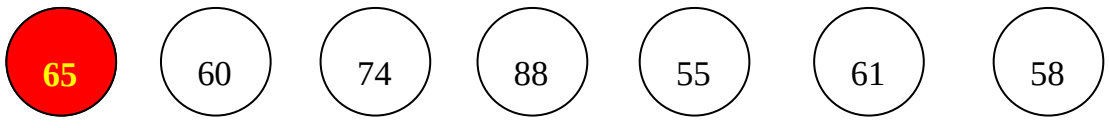
Thuật toán Quicksort còn gọi là sắp xếp theo kiểu phân đoạn (partition sort).

Chọn một phần tử bất kỳ làm chốt (pivot). Khi một phần tử được chọn làm chốt thì mọi phần tử nhỏ hơn chốt được chuyển về phía trước chốt, mọi phần tử lớn hơn chốt sẽ được đẩy về phía sau chốt. Cuối cùng các phần tử nhỏ hơn chốt sẽ tạo thành một mảng con thứ nhất, các phần tử lớn hơn chốt sẽ tạo thành mảng con thứ hai. Vị trí nằm giữa hai mảng này chính là giá trị chốt mà ta vừa chọn. Thực hiện tương tự cho hai dãy con vừa tạo ra cho đến khi dãy được sắp xếp hoàn toàn.

Ví dụ: Xét mảng A có các phần tử như sau:



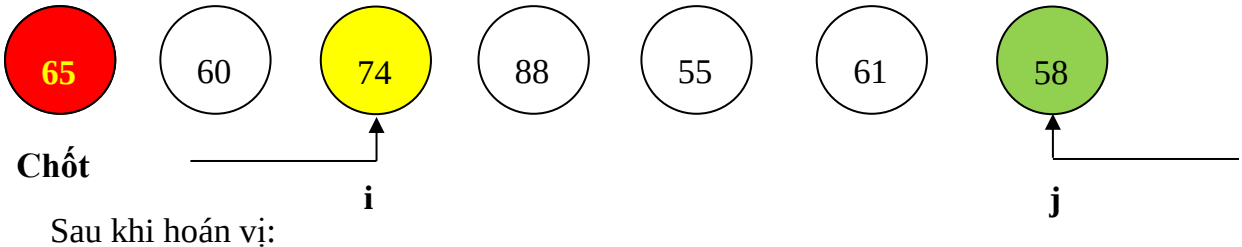
Giả sử chọn phần tử đầu tiên (65) để làm chốt. chuyển các số nhỏ hơn 65 là 60, 55, 61, 58 về phía bên trái (phía trước) của số 65 và chuyển các số lớn hơn 65 là 74 và 88 về phía bên phải (phía sau) của 65.



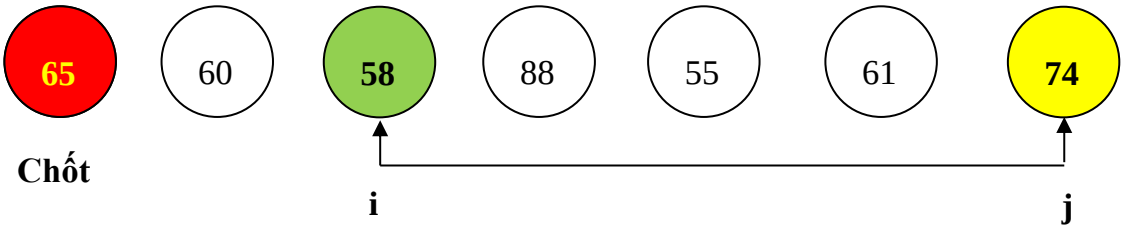
Chốt

Để thực hiện việc tách mảng A thành hai mảng con như đã nêu, ta thực hiện hai phép tìm kiếm ngược chiều nhau: Một phép tìm kiếm bắt đầu từ phần tử thứ hai bên trái (vì

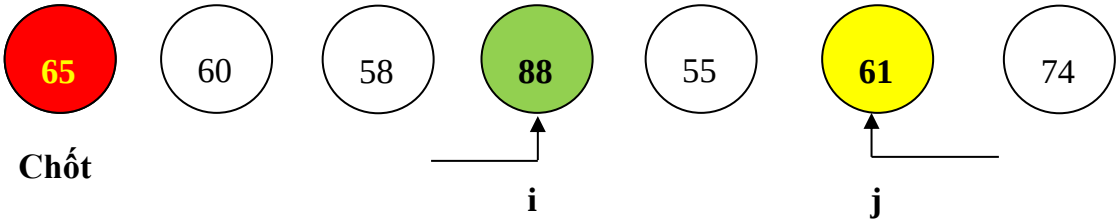
phần tử thứ nhất đã được ta chọn làm chốt rồi) để truy ra các phần tử lớn hơn chốt, và một phép tìm kiếm ngược lại bắt đầu từ phần tử ngoài cùng bên phải để truy ra những phần tử nhỏ hơn chốt. Chỉ số của phần tử truy tìm từ trái được ghi nhận bởi biến **i** (ở đây i = 3 ứng với phần tử có giá trị là 74), chỉ số của phần tử truy tìm từ phải được ghi nhận bởi biến **j** (j = 7 ứng với phần tử có giá trị là 58). Ta thấy i < j vì vậy hai phần tử 74 và 58 được chuyển chỗ cho nhau. Hình minh họa như sau:



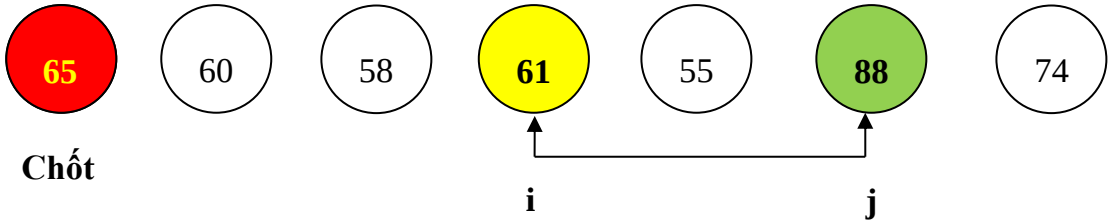
Sau khi hoán vị:



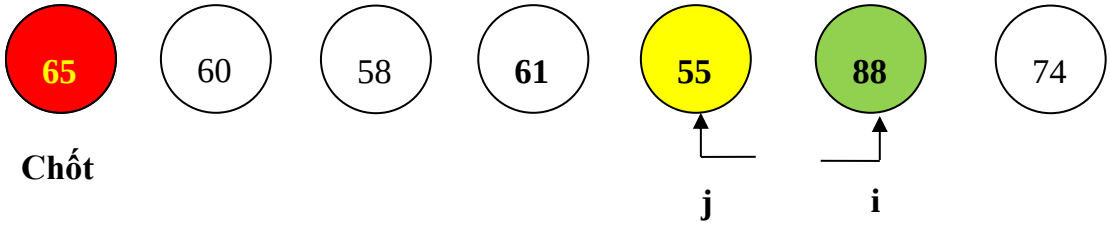
Tiếp tục quá trình trên, từ bên trái thì i sẽ ghi nhận: i = 4 ứng với phần tử 88, và từ bên phải thì j ghi nhận: j=6 ứng với phần tử 61. Lúc này i<j, hai phần tử trên lại được đổi chỗ cho nhau. Hình minh họa như sau:

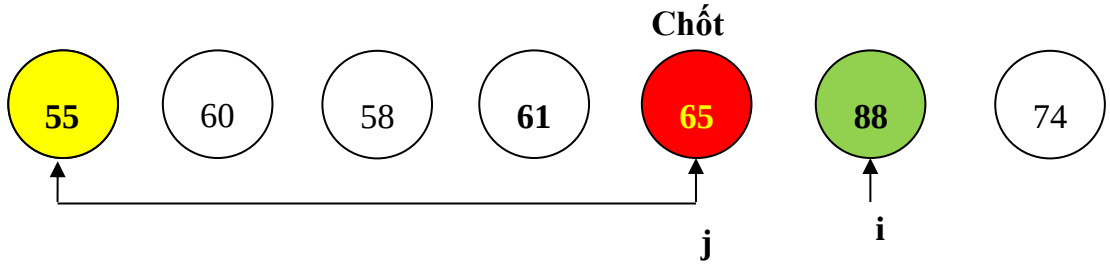


Sau khi hoán vị:



Lại tiếp tục từ bên trái ta có i = 6 ứng với phần tử 88, còn từ bên phải thì cuối cùng j = 5 ứng với phần tử 55. Lúc này i>j thì việc hoán vị được thực hiện giữa 55 và chốt 65. Hình minh họa như sau:

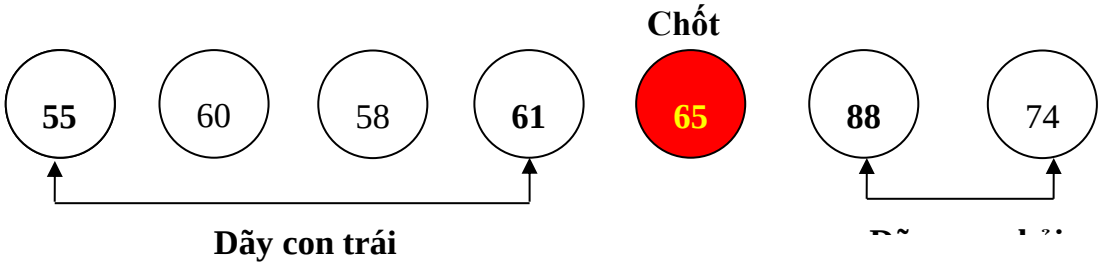




Như vậy sau khi đã duyệt xong hai vòng lặp ở lượt thứ nhất thì dãy được chia thành hai dãy con: dãy con trái chứa các số nhỏ hơn chốt ở đây là 65 và dãy con phải chứa số lớn hơn chốt ở đây là 65. Giải thuật lại tiếp tục với các công việc tương tự cho lần lượt hai dãy con vừa tạo ra là dãy con trái và dãy con phải, và được gọi đệ quy cho các dãy con tạo ra với kích thước dữ liệu nhỏ dần cho đến khi mảng được sắp xếp hoàn toàn thì dừng. Lưu ý là lời gọi đệ quy trong phạm vi của dãy con trái từ đầu mảng cho đến phần tử có chỉ số nhỏ hơn chỉ số của vị trí chốt đã cho là 1 đơn vị, và lời gọi đệ quy trong phạm vi của dãy con phải từ phần tử có chỉ số lớn hơn chỉ số của vị trí chốt đã cho là 1 đơn vị, vị trí chốt đã được đặt đúng vị trí của nó trong dãy sắp xếp cuối cùng nên chúng ta không tính đến nữa.

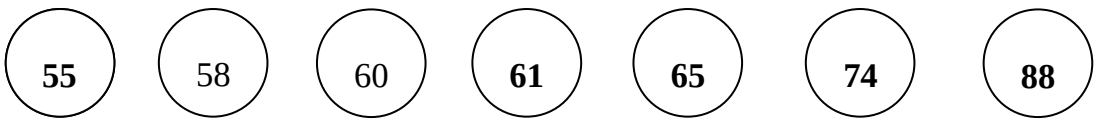
Với ví dụ trên ta có thể thấy sau một lượt sắp xếp thì đã chia thành hai mảng, mảng con trái gồm các phần tử có giá trị nhỏ hơn chốt là 55, 60, 58, 61 và dãy con phải gồm các phần tử 88, 74 với chốt là 65. Quá trình lại tiếp tục với các mảng con bằng một kỹ thuật tương tự. Ví dụ với mảng con trái gồm các phần tử 55, 60, 58, 61 thì ta có quá trình sắp xếp diễn ra như sau:

Chọn phần tử đầu tiên có giá trị là 55 làm chốt của dãy con trái, lúc đó ta sẽ áp dụng thuật toán như ban đầu để chia dãy con trái thành hai dãy con của nó, một dãy phía trước chứa các giá trị nhỏ hơn 55, một dãy phía sau chứa các giá trị lớn hơn 55.



Khi i =4 ứng với phần tử 61 và j =4 cũng ngay phần tử có giá trị 61, lúc này i=j nên giá trị tại vị trí i và phần tử tại vị trí j chuyển chỗ cho nhau và nó sẽ chia mảng con trái này thành 2 mảng nhỏ hơn, mảng con trái con không chứa phần tử nào và mảng con phải con các phần tử lớn hơn 55 là 60, 58, 61 phần tử chốt 55 là đã đúng vị trí.

Và nó lại gọi đệ quy tiếp cho hai mảng con vừa tạo ra cho đến khi mảng được sắp xếp hoàn toàn như sau:



Như vậy ta có thể mô tả tổng quát về phương pháp Quicksort như sau:

Để sắp xếp mảng a[l]..a[r] ta tiến hành các bước:

- Xác định chốt.
- Phân hoạch mảng đã cho thành hai mảng con a[i]..a[k-1] và a[k]..a[j].
- Sắp xếp mảng a[l]..a[k-1] (Đệ quy).
- Sắp xếp mảng a[k+1]..a[r] (Đệ quy).

Quá trình đệ quy sẽ dừng khi không còn tìm thấy chốt.

Giải thuật mô tả bằng ngôn ngữ C

```
void Quick_Sort(int left,int right,int A[])
{
  int pivot = A[(left+right)/2];
  int i = left, j = right;
  while(i<j)
  {
   while(A[i] < pivot) i++;
   while(A[j] > pivot) j--;
   if(i<=j)
   {
     Swap(A[i],A[j]);
     i++; j--;
   }
  }
  if(left<j) Quick_Sort(left,j,A);
  if(right>i) Quick_Sort(i,right,A);
}
```

2.5.2. Thuật toán tìm kiếm

2.5.2.1. Tìm kiếm tuần tự

Đây là một kỹ thuật tìm kiếm cổ điển. Thuật toán tiến hành so sánh x lần lượt với phần tử thứ nhất, thứ hai, ... của mảng a cho đến khi gặp được phần tử có khóa cần tìm, hoặc duyệt tìm hết mảng mà không thấy x.

```
int Search(int x, int n, int a[])
{
     int i=0;
```

```
    while ((i<=n) && (a[i] != X)) i++;
    if(i>n)return 0;   //không tìm thấy
    else return i;     //vị trí tìm thấy
}
```

2.5.2.2. Tìm kiếm nhị phân

Áp dụng đối với mảng đã được sắp xếp theo thứ tự tăng hoặc giảm dần.

```
int Tim(int x,int dau,int cuoi,int a[])
{
  if(dau>cuoi) return 0;
  else
     {
       int giua = (dau + cuoi)/2;
       if(x==a[giua]) return giua+1;
       else if(x<a[giua]) return Tim(x,dau,giua-1,a);
            else return Tim(x,giua+1,cuoi,a);
     }
}
```

BÀI TẬP

Bài 1: (ARRMAX) Viết chương trình tìm giá trị lớn nhất của một mảng số nguyên gồm N phần tử.

Input:

- Dòng đầu chứa số nguyên: N
- Dòng tiếp theo chứa các phần tử a1,…,an

Output: M là số nguyên lớn nhất trong mảng

Ví dụ:

INPUT	OUTPUT
5 1 2 30 4 5	30

Bài 2: (MERGEARR) Cho 2 mảng số nguyên đã được sắp xếp theo thứ tự tăng dần. Hãy trộn 2 mảng đó lại thành mảng C sao cho mảng C vẫn có thứ tự tăng dần.

Input:

- Dòng đầu chứa 2 số nguyên: M, N ≤ 500,000
- Dòng tiếp theo chứa các phần tử $a_1,\ldots,a_M$ tăng dần

- Dòng tiếp theo chứa các phần tử $b_1,\ldots,b_N$ tăng dần ($a_i, b_i \le 10^6$)

Output:

- Dòng đầu ghi: M+N
- Dòng sau ghi: $C_1, C_2,\ldots, C_{M+N}$ tăng dần

Ví dụ:

INPUT	OUTPUT
5 3 1 3 5 7 9 2 4 6	8 1 2 3 4 5 6 7 9

Gợi ý:

- Dùng 2 chỉ số i,j để duyệt qua các phần tử của 2 mảng A, B và k là chỉ số cho mảng C.

- Trong khi (i<m) và (j<n) thì:

*/*Tức trong khi cả 2 dãy A, B đều chưa duyệt hết */*

+ Nếu A[i]>B[j] thì: C[k]=A[i]; i=i+1;

+ Ngược lại: C[k]=B[j]; j=j+1;

- Nếu dãy nào hết trước thì đem phần còn lại của dãy kia bổ sung vào cuối dãy C.

Bài 3: (DAYTANG) Viết chương trình nhập vào một dãy số nguyên $a_1, a_2, ..., a_n$. Tìm độ dài dãy con tăng dần dài nhất. (Dãy con là dãy các phần tử liên tiếp nhau trong mảng).

Input:

- Dòng đầu chứa số nguyên: N ($N \le 10^6$)
- Dòng tiếp theo chứa các phần tử $a_1,\ldots,a_N$ ($a_i \le 10^{12}$)

Output: S là độ dài dãy con tăng dần dài nhất

Ví dụ:

INPUT	OUTPUT
10 9 12 **2 2 3 7 8** 5 10 15	5

Bài 4: (SETNUM) Cho *n* số nguyên dương $a_1, a_2,..., a_n$ ($1 \le n, a_i \le 10^6$). Hãy xác định số lượng các phần tử khác nhau trong dãy số đã cho.

Input:

- Dòng đầu chứa số nguyên dương n.
- Dòng sau chứa n số nguyên $a_1, a_2,..., a_n$.

Output: một số nguyên S là số các phần tử khác nhau trong dãy số đã cho.

Ví dụ:

INPUT	OUTPUT
5 4 1 4 2 1	3

Bài 5: (SUPASCEN) Dãy số nguyên dương được gọi là dãy ***siêu tăng*** nếu kể từ phần tử thứ hai trở đi, mỗi phần tử không nhỏ hơn tổng của các phần tử đứng trước đó.

Ví dụ: a = {1, 5, 6, 15, 30} là dãy siêu tăng; b = {1, 5, 9, 10, 21} không phải là dãy siêu tăng.

Cho dãy số nguyên dương {a} gồm n phần tử ($n \leq 50$, $a < 10^{18}$). Hãy kiểm tra tính siêu tăng của dãy a.

Input:

- Dòng đầu chứa số nguyên dương n.
- Dòng sau chứa n số nguyên $a_1, a_2, ..., a_n$.

Output: TRUE/FALSE ứng với dãy siêu tăng.

Ví dụ:

INPUT	OUTPUT
5 1 5 6 15 30	TRUE

INPUT	OUTPUT
5 1 5 9 10 21	FALSE

Bài 6: (SYMABILITY) Dãy số nguyên được gọi là dãy ***khả đối xứng*** nếu có thể sắp xếp lại thành một dãy đối xứng. Cho dãy số nguyên gồm n phần tử (n>0). Hãy kiểm tra dãy có phải là khả đối xứng?

Input:

- Dòng đầu: n ($n \leq 10^6$)
- Dòng sau ghi n số nguyên $a_1, a_2, ..., a_n$ ($a_i \leq 10^{18}$).

Output: TRUE/FALSE tương ứng với khả đối xứng.

Ví dụ:

INPUT	OUTPUT
5 1 5 6 5 6	TRUE

Bài 7: (PASCAL) Viết chương trình in ra màn hình tam giác Pascal.

Input: $N \leq 50$

Output: Ma trận tam giác Pascal

Ví dụ:

INPUT	OUTPUT
4	1 1 1 1 2 1 1 3 3 1 1 4 6 4 1

Ý tưởng:

Tam giác Pascal được tạo ra theo qui luật sau:

+ Mỗi dòng đều bắt đầu và kết thúc bởi số 1.

+ Phần tử thứ j ở dòng i nhận được bằng cách cộng 2 phần tử thứ j-1 và j ở dòng thứ i-1: **a(i,j) = a(i-1,j-1) + a(i-1,j)**.

Bài 8: (FSEQ) (Olympic Tin học 2013, khối Cao đẳng)

Một dãy số gồm ***n*** số nguyên $f_1,f_2,\ldots,f_n$ được gọi là dãy có tính chất của dãy số Fibonacci nếu $n\geq 3$ và với mọi số f_i ($i\geq 3$) thỏa mãn điều kiện $f_i = f_{i-1} + f_{i-2}$.

Ví dụ, dãy 1, 1, 2, 3, 5, 8 là dãy số có tính chất của dãy số Fibonacci; còn dãy 3, 3, 6, 9, 14**,** 23 không phải là dãy số có tính chất của dãy số Fibonacci.

Yêu cầu: Cho dãy số nguyên $a_1, a_2,\ldots,a_n$. Hãy tìm một dãy con liên tiếp gồm nhiều phần tử nhất của dãy số đã cho có tính chất của dãy số Fibonacci.

Input*:*

- Dòng đầu chứa số nguyên n ($3 \leq n \leq 30000$)

- Dòng thứ hai chứa n số nguyên $a_1, a_2,\ldots,a_n$ ($|a_i| \leq 10^9$).

Output*:* Một số nguyên là số lượng phần tử của dãy con tìm được, ghi -1 nếu không tồn tại dãy con liên tiếp nào của dãy có tính chất của dãy số Fibonacci.

Ví dụ:

INPUT	OUTPUT
7 1 **3 3 6 9** 14 23	4

Bài 9: Viết chương trình tính tổng của 2 đa thức h(x) = f(x) + g(x). Trong đó, mỗi đa thức có dạng: $a_0 + a_1x + a_2x^2 + ... + a_nx^n$.

Gợi ý:

Dùng các mảng A, B, C để lưu trữ các hệ số a_i của các đa thức f(x), g(x) và h(x).

Bài 10: Viết chương trình nhập vào số tự nhiên N (N lẻ), sau đó điền các số từ 1 đến n^2 vào trong một bảng vuông sao cho tổng các hàng ngang, hàng dọc và 2 đường chéo đều bằng nhau (bảng này được gọi là Ma phương).

Ví dụ: Với N=3 và N=5 ta có

2	7	6
9	5	1
4	3	8

Bắc

	3	16	9	22	15
	20	8	21	14	2
Tây	7	25	13	1	19	Đông
	24	12	5	18	6
	11	4	17	10	23

Nam

Phương pháp:

Xuất phát từ ô bên phải của ô giữa. Đi theo ***hướng đông bắc*** để điền các số 1, 2, ...

Khi điền số, cần chú ý một số nguyên tắc sau:

- Nếu vượt ra phía ngoài bên phải của bảng thì quay trở lại cột đầu tiên.
- Nếu vượt ra phía ngoài bên trên của bảng thì quay trở lại dòng cuối cùng.
- Nếu số đã điền k chia hết cho N thì số tiếp theo sẽ được viết trên cùng một hàng với k nhưng cách 1 ô về phía bên phải.

Bài 11: Viết chương trình in ra các số nguyên từ 1 đến N^2 theo hình xoắn ốc với N được nhập vào từ bàn phím. Ví dụ, với N=5 ta có:

1	2	3	4	5
16	17	18	19	6
15	24	25	20	7
14	23	22	21	8
13	12	11	10	9

Bài 12: Tìm hiểu và viết chương trình cài đặt các thuật toán sắp xếp:

1. Heap sort
2. Merge sort
3. Radix sort
4. Shell sort

CHƯƠNG III: CẤU TRÚC DỮ LIỆU ĐỘNG

3.1. DANH SÁCH LIÊN KẾT

3.1.1. Định nghĩa

Danh sách liên kết (linked list) là một danh sách mà các phần tử được nối kết với nhau nhờ vào vùng liên kết của chúng.

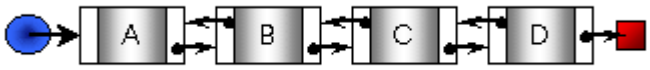
Danh sách liên kết là một loại cấu trúc đơn giản và thích hợp với các phép thêm vào, phép loại bỏ, phép ghép nhiều danh sách mà các phép toán này lại không thích hợp cho danh sách đặc.

Để tổ chức một danh sách mà số phần tử của dãy luôn biến động (tức là thao tác thêm vào và loại bỏ được dùng nhiều) ta dùng danh sách liên kết mà các phần tử của nó có thể được chứa ở những vùng không kế cận nhau trong bộ nhớ và chúng được nối với nhau nhờ vào vùng liên kết: vùng liên kết của phần tử thứ nhất chứa địa chỉ của phần tử thứ hai, vùng liên kết của phần tử thứ hai chứa địa chỉ của phần tử thứ ba và cứ như thế cho đến phần tử cuối cùng, vùng liên kết của phần tử cuối cùng là NULL.

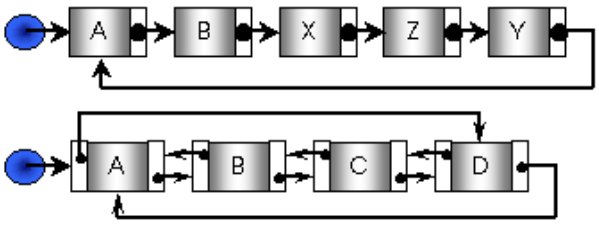
3.1.2. Tổ chức danh sách

Danh sách liên kết đơn: mỗi phần tử liên kết với phần tử đứng sau nó trong danh sách:

Danh sách liên kết kép: mỗi phần tử liên kết với các phần tử đứng trước và sau nó trong danh sách:



Danh sách liên kết vòng: phần tử cuối danh sách liên kết với phần tử đầu danh sách:



Hình thức liên kết này cho phép các thao tác chèn, xóa trên danh sách được thực hiện dễ dàng, phản ánh được bản chất linh động của danh sách.

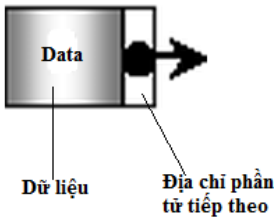
3.2. DANH SÁCH LIÊN KẾT ĐƠN

3.2.1. Định nghĩa và khai báo

Danh sách liên kết đơn là danh sách liên kết mà mỗi phần tử (Node) của nó chỉ có một trường liên kết chứa địa chỉ của phần tử (Node) đứng ngay sau nó.

Mỗi phần tử của danh sách liên kết đơn là một cấu trúc chứa 2 thông tin:

- Thành phần dữ liệu: lưu trữ các thông tin về bản thân phần tử .
- Thành phần liên kết: lưu trữ địa chỉ của phần tử kế tiếp trong danh sách, hoặc lưu trữ giá trị NULL nếu là phần tử cuối danh sách. Xem hình minh họa sau:



Để định nghĩa một danh sách liên kết trước hết phải định nghĩa kiểu của mỗi **nút** trong danh sách.

```
struct <NUT>
{
  <Kiểu_dữ_liệu> <Data>;// chứa dữ liệu
  NUT *next;            // con trỏ trỏ đến nút tiếp theo
};
typedef NUT* <Trỏ_Nút>;
```

Sau đó có thể khai báo một danh sách liên kết như sau:

<Trỏ_Nút> First;

First là địa chỉ của nút đầu tiên trong danh sách, dựa vào trường **next** của nút này để biết được địa chỉ của các nút tiếp theo trong danh sách.

Ví dụ: Tạo một danh sách liên kết chứa các số nguyên:

```
struct NUT
{
  int x;                          // chứa số nguyên
     NUT *next;
};
typedef NUT* TroDS;
TroDS First; //Danh sách được trỏ bởi con trỏ đầu First
```

3.2.2. Các thao tác trên danh sách liên kết đơn

3.2.2.1. Khởi tạo danh sách

```
First=NULL;
```

3.2.2.2. Bổ sung một phần tử vào danh sách

Trường hợp 1: Bổ sung vào đầu danh sách

Bước 1: Tạo ra nút mới

```
<Trỏ_Nút> p=new(NUT);
p->Data = <Giá trị>;
```

Bước 2: Bổ sung vào đầu danh sách

```
p->next=First;
First=p;
```

Trường hợp 2: Bổ sung vào cuối danh sách

```
void BSCuoi(int x,TroDS &First)
{
 TroDS p,q;
 //Tao nut moi
 p=new(NUT);
 p->x=x;
 p->next=NULL;

 if(First==NULL) First=p;
 else
 {
  //Tim den nut cuoi
  q=First;
  while(q->next!=NULL) q=q->next;
  //Bo sung
  q->next=p;
 }
}
```

Trường hợp 3: Bổ sung vào sau node được trỏ bởi p

Hàm sau thực hiện việc bổ sung một nút mới có nội dung x vào sau nút được trỏ bởi p.

```
void Bosung(int x,TroDS p,TroDS &First)
{
 //Tạo nút mới q
 TroDS q = new(NUT);
 q->x=x;

 //Bổ sung
 if(First==NULL)  //Danh sach rong
 {
    q->next=NULL;
    First=q;
 }
 else
    {
      //Bổ sung nút q vào sau nút p
      q->next=p->next;
      p->next=q;
    }
}
```

3.2.2.3. Loại bỏ một node khỏi danh sách

Lưu ý rằng khi cấp phát bộ nhớ, chúng ta đã dùng phương thức ***new***. Vì vậy khi giải phóng bộ nhớ ta phải dùng phương thức ***delete***.

Trường hợp 1: Loại bỏ node đầu tiên của danh sách

```
void XoaDau(TroDS &First)
{
 TroDS p = First;
 First=First->next;
 delete(p);
}
```

Trường hợp 2: Loại bỏ node đứng sau node được trỏ bởi q

```
void Xoa(TroDS q,TroDS &First)
{
  if(p->next)
```

```
  {
     TroDS p = q->next;
     q->next = p->next;
     delete(p);
  }
}
```

3.2.2.4. Duyệt danh sách

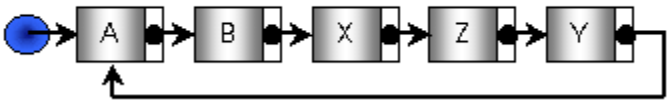
Duyệt qua và xử lý từng nút trong danh sách.

```
void DuyetDS(TroDS First)
{
  TroDS p;
  p=First;
  while(p!=NULL)
  {
    <Xử lý p->x>;
    p=p->next;
  }
}
```

3.2.3. Danh sách liên kết đơn nối vòng

3.2.3.1. Định nghĩa

Danh sách liên kết đơn nối vòng là một danh sách liên kết đơn mà phần tử cuối cùng chỉ vào (có liên kết với) phần tử đầu tiên của danh sách.



3.2.3.2. Các thao tác trên danh sách liên kết đơn nối vòng

a. Khởi tạo danh sách liên kết đơn nối vòng

Tương tự như khởi tạo danh sách liên kết đơn nhưng khi đến phần tử cuối cùng ta cho con trỏ last trỏ vào phần tử đầu tiên của danh sách.

b. Tìm một phần tử trên danh sách liên kết đơn nối vòng

Danh sách liên kết đơn nối vòng không có phần tử đầu danh sách rõ rệt, nhưng ta có thể đánh dấu một phần tử bất kỳ trên danh sách xem như phần tử đầu danh sách để kiểm tra việc duyệt đã qua hết các phần tử của danh sách hay chưa.

c. Thêm một phần tử vào danh sách

- **Trường hợp 1: Thêm một node vào đầu danh sách**

Danh sách liên kết đơn nối vòng không có phần tử đầu danh sách rõ rệt, nhưng ta có thể đánh dấu một phần tử bất kỳ trên danh sách xem như phần tử đầu danh sách để thực hiện thêm phần tử vào đầu danh sách.

Thuật toán:

Bước 1: Tạo nút mới cần thêm vào là p;

Bước 2: Nếu danh sách rỗng thì thực hiện

```
First=p;
First->Next = First;
```

Ngược lại: Chuyển qua bước 3.

Bước 3: Tìm đến nút cuối cùng của danh sách: q

```
Tro q = First;
While (q->Next !=First) q = q->Next;
p->Next = First;
First = p;
q->Next = First;
```

- **Trường hợp 2: Thêm một node vào cuối danh sách**

Thuật toán:

Bước 1: Tạo nút mới e;

Bước 2: Nếu danh sách rỗng thì thực hiện

```
First = e;
First->Next = First;
```

Ngược lại: Chuyển qua bước 3.

Bước 3: Tìm đến nút cuối cùng của danh sách

```
Tro p = First;
While (p->Next !=First) p=p->Next;
e->Next = pHead;
p->Next=e;
```

- **Trường hợp 3: Thêm một node vào sau một node được trỏ bởi q (tương tự trường hợp danh sách móc nối đơn bình thường).**

d. Xóa một phần tử trong danh sách

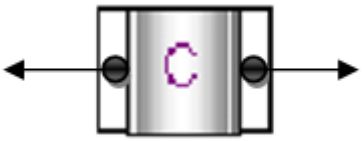
- **Trường hợp 1: Hủy node đầu danh sách**
- **Trường hợp 2: Hủy một node đứng sau node q**

3.3. DANH SÁCH LIÊN KẾT KÉP

3.3.1. Định nghĩa

Danh sách liên kết kép là một danh sách liên kết mà mỗi phần tử có hai vùng liên kết: một vùng liên kết chỉ đến phần tử đứng ngay trước nó, gọi là liên kết ngược (Previous) và một vùng liên kết chỉ đến phần tử đứng ngay sau nó gọi là liên kết thuận(Next). Hình minh họa cho danh sách liên kết kép như sau:

Một phần tử của danh sách liên kết kép có dạng:



Ta có định nghĩa tổng quát:

```
struct Node {
        <Kiểu_dữ_liệu> Data;
        Node *Pre;//con trỏ chỉ đến node liền trước
        Node  *Next;//con trỏ chỉ đến cấu trúc node tiếp theo
        };
typedef Node* Tro;
```

3.3.2. Các thao tác trên danh sách liên kết kép

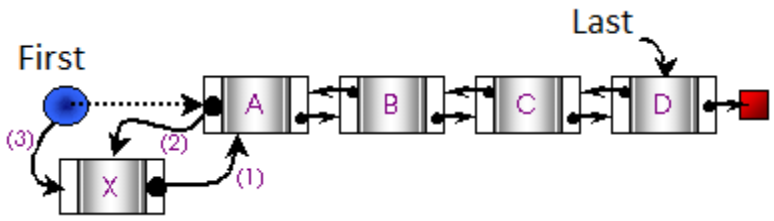
3.3.2.1. Khởi tạo một danh sách liên kết kép

Khi khởi tạo danh sách liên kết kép ta khai báo hai biến con trỏ First và Last trỏ đến đầu và cuối danh sách:

Tro First = Last = NULL;

3.3.2.2. Bổ sung một phần tử vào danh sách

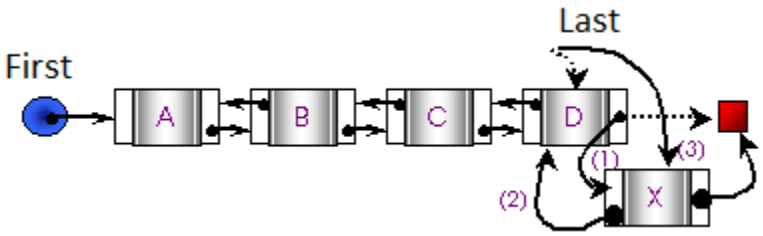
➢ **Trường hợp 1: Chèn Node vào đầu danh sách**




```
void ChenDau(int x,Tro &First,Tro &Last)
{
  Tro p = new(NUT);
```

```
  p->Data = x;
  p->Pre = NULL;
  if(First==NULL)
  {
    p->Next = NULL;
    First=Last=p;
  }
  else
     {
       p->Next = First;
       First->Pre = p;
       First = p;
     }
}
```

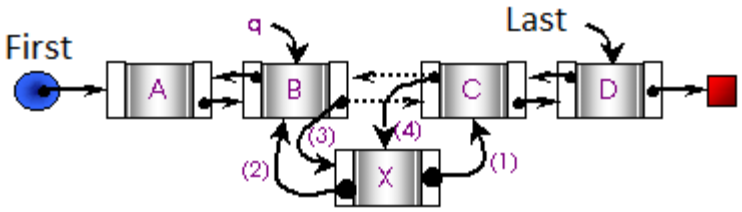
➢ **Trường hợp 2:Chèn vào cuối danh sách**




```
void ChenCuoi(int x,Tro &First,Tro &Last)
{
  Tro p = new(NUT);
  p->Data = x;
  p->Next = NULL;
  if(First==NULL)
  {
    p->Pre = NULL;
    First=Last=p;
  }
  else
     {
       p->Pre = Last;
       Last->Next = p;
```

```
        Last = p;
      }
}
```

➢ **Trường hợp 3: Chèn vào sau node được trỏ bởi q**




```
void Chen(int x,Tro q,Tro &First,Tro &Last)
{
  Tro p = new(NUT);
  p->Data = x;
  p->Next = q->Next;
  p->Pre = q;
  q->Next->Pre = p;
  q->Next = p;
}
```

3.3.2.3. Loại bỏ một phần tử ra khỏi danh sách

➢**Trường hợp 1: Loại bỏ Node đầu danh sách**

```
void XoaDau(Tro &First,Tro &Last)
{
  Tro p = First;
  if(First==Last) First=Last=NULL;
  else
     {
          First = p->Next;
          First->Pre = NULL;
     }
  delete(p);
}
```

➢**Trường hợp 2: Loại bỏ Node cuối danh sách**

```
void XoaCuoi(Tro &First,Tro &Last)
```

```
{
  Tro p = Last;
  if(First==Last) First=Last=NULL;
  else
     {
          Last = p->Pre;
          Last->Next = NULL;
     }
  delete(p);
}
```

➢ **Trường hợp 3: Loại bỏ Node đứng sau Node được trỏ bởi q**

```
void XoaGiua(Tro q,Tro &Dau,Tro &Cuoi)
{
  Tro p = q->Next;
  if(p!=NULL)
  {
    q->Next = p->Next;
    p->Next->Pre = q;
    delete(p);
  }
}
```

BÀI TẬP

Bài 1: Cho danh sách các số nguyên được tổ chức như sau:

```
struct PhanTu
{
  int x;
  PhanTu *next;
};
typedef PhanTu* TRO;
```

a/ Viết hàm **void NHAP(TRO &Dau)** để nhập vào một danh sách các số nguyên có nút đầu tiên được trỏ bởi con trỏ Dau.

b/ Viết hàm **void LIETKE(TRO Dau)** để in ra màn hình giá trị của tất cả các nút trong danh sách được trỏ bởi con trỏ Dau.

c/ Viết hàm **int MAX(TRO Dau)** để tìm đến nút có giá trị lớn nhất trong danh sách được trỏ bởi con trỏ Dau, nếu danh dách rỗng thì hàm trả về gá trị NIL.

d/ Viết hàm **int DEM(TRO Dau)** để đếm xem trong danh sách có bao nhiêu nút.

e/ Viết hàm **float TBINH(TRO Dau)** để tính giá trị trung bình của các phần tử trong danh sách.

f/ Viết hàm **SAPXEP(TRO &L)** để sắp xếp lại danh sách L theo thứ tự tăng dần.

g/ Viết hàm **TRO CUOI(TRO L)** để xác định nút cuối cùng của danh sách L.

h/ Viết hàm **void NOI(TRO P, TRO &L)** để nối nút P cho trước vào cuối của danh sách L.

i/ Viết hàm **void XOACUOI(TRO &L);** để xoá nút cuối cùng trong danh sách được trỏ bởi con trỏ L.

j/ Viết hàm **COPY(TRO L, TRO &Dau);** để sao chép các nút có trường x chẵn trong danh sách được trỏ bởi con trỏ L sang danh sách mới được trỏ bởi con trỏ Dau.

k/ Giả sử Dau là con trỏ trỏ đến đầu của một danh sách chưa được sắp xếp, còn L được gán bằng NIL. Hãy viết hàm **SAPCHON(TRO &Dau, TRO &L)** cho phép chọn các phần tử trong danh sách Dau từ giá trị lớn đến bé, đưa vào danh sách L để cuối cùng Dau=Nil, còn L sẽ trở thành một danh sách đã được sắp xếp theo thứ tự tăng dần.

l/ Viết hàm **int TANG(TRO Dau)** để xác định xem danh sách được trỏ bởi Dau đã có thứ tự tăng dần hay không theo 2 cách: ***Không đệ qui*** và ***đệ qui***.

m/ Viết hàm **void DAO(TRO &Dau)** để đảo ngược các con trỏ của danh sách Dau.

Bài 2: Cho 2 danh sách liên kết biểu diễn cho 2 tập hợp các số nguyên được trỏ bởi L1 và L2.

a/ Viết hàm **GIAO(L1,L2);** để hiển thị phần giao của 2 danh sách trên.

b/ Viết hàm **HOP(L1,L2);** để hiển thị phần hợp của 2 danh sách trên.

c/ Viết hàm **HIEU(L1,L2);** để hiển thị phần hiệu của 2 danh sách trên.

Bài 3: Cho 2 danh sách móc nối được trỏ bởi L1 và L2. Hãy viết một thủ tục để nối 2 danh sách đó lại thành một danh sách được trỏ bởi L1.

Bài 4: Cho một danh sách móc nối các số nguyên. Hãy viết thủ tục để xóa tất cả các nút chứa các giá trị âm trong danh sách đó.

Bài 5: Dùng danh sách móc nối đơn để biểu diễn một đa thức $\mathbf{P_n(x) = a_n x^n + a_{n-1} x^{n-1} + ... + a_0}$. Trong đó, mỗi số hạng của đa thức được xác định bởi 2 thành phần: hệ số $\mathbf{a_i}$ và số mũ **i**. Ta có thể xây dựng cấu trúc dữ liệu để chứa đa thức như sau:

```
struct Node
{
        float HeSo;
        int SoMu;
        Node *Next;
```

```
}
```

typedef Node* DATHUC;

Thực hiện các công việc sau:

1. Viết hàm nhập vào một đa thức.

2. Viết hàm để sắp xếp lại các số hạng của đa thức theo thứ tự số mũ giảm dần.

3. Viết hàm để cộng hai đa thức S1, S2 thành đa thức S3.

4. Viết hàm để tính giá trị của đa thức theo giá trị X cho trước.

Bài 6: Cho một file văn bản trong đó có chứa các từ. Các dấu phân cách từ là: ký tự trắng, dấu chấm, dấu phẩy, dấu chấm phẩy, dấu hai chấm, dấu than, dấu hỏi. Mọi từ đều bắt đầu bằng một ký tự trong tập ['A'..'Z']. Ta tổ chức từ điển bằng mảng các danh sách móc nối như sau:

```
struct Nut
{
     char Tu[10];
     Nut *Next;
}
typedef Nut* DanhSach;
DanhSach TuDien[26];  //26 chữ cái từ 'A' đến 'Z'
```

Mỗi danh sách móc nối trong từ điển đều phải được sắp thứ tự (tăng dần), và các từ được lưu trong từ điển phải khác nhau.

1. Viết hàm bổ sung một từ mới vào từ điển bằng cách đọc từ đó từ bàn phím:
   - Tìm từ đó trong từ điển.
   - Nếu tìm thấy, hiển thị thông báo:"Từ đã có trong từ điển".
   - Nếu không tìm thấy, chèn từ đó vào trong từ điển ở vị trí thích hợp.

2. Viết hàm để đọc các từ trong file văn bản và lưu các từ đó vào từ điển.

3. Viết hàm hiển thị tất cả các từ trong từ điển ra màn hình theo thứ tự tăng dần.

Bài 7: Cho danh sách các số nguyên được tổ chức như sau:

```
struct Nut
{
  int x;
  Nut *next;
};
typedef Nut* TRO;
```

1. Viết hàm **int DXUNG(TRO ds)** nhằm kiểm tra danh dách ds có phải là đối xứng hay không? (Viết theo 2 cách: Đệ qui và không đệ qui)

2. Viết hàm **int KhaDX(TRO ds)** để kiểm tra xem với danh sách ds đã cho, có tồn tại một cách sắp xếp lại các phần tử để cuối cùng nhận được một danh sách đối xứng hay không?

3. Viết hàm **void SAPLAI(TRO &ds)** để đưa ra một cách sắp xếp lại các phần tử để có một danh sách đối xứng (giả thiết điều này có thể làm được nhờ hàm **KhaDX**).

Bài 8: Dùng danh sách móc nối đơn để biểu diễn một đa thức $\mathbf{P_n(x) = a_n x^n + a_{n-1}x^{n-1} +...+ a_0}$.

Thực hiện các công việc sau:

1. Viết thủ tục nhập vào một đa thức.

2. Viết thủ tục để sắp xếp lại các số hạng của đa thức theo thứ tự số mũ giảm dần.

3. Viết hàm để cộng 2 đa thức S1, S2 thành đa thức S3.

4. Viết hàm để tính giá trị của đa thức theo giá trị X cho trước.

Bài 9: Cho một danh sách liên kết kép chứa một dãy số nguyên. Viết chương trình thực hiện các phép toán sau:

1. Viết hàm nhập vào n Node từ bàn phím.
2. Viết hàm in danh sách ra màn hình theo chiều xuôi và chiều ngược của danh sách.
3. Đếm xem trong danh sách có bao nhiêu phần tử chẵn, có bao nhiêu phần tử lẻ, có bao nhiêu phần tử âm và có bao nhiêu không âm?
4. Kiểm tra danh sách có đối xứng hay không?

Bài 10: QUẢN LÝ KHÁCH HÀNG

Để quản lý khách hàng sử dụng điện thoại SmartPhone, người ta tổ chức lưu trữ danh sách khách hàng dưới dạng danh sách liên kết đơn với 4 trường sau: ***Name*** chứa họ tên khách hàng, 2 con trỏ ***First*** và ***Last*** lần lượt trỏ tới phần tử đầu tiên và cuối cùng của một danh sách liên kết đơn khác dùng để ghi nhận danh sách điện thoại của khách hàng đó đang sở hữu, cuối cùng là trường ***Next*** trỏ đến khách hàng tiếp theo. Mỗi phần tử của danh sách liên kết chứa điện thoại là một cấu trúc gồm 3 trường: ***PhoneName*** (tên điện thoại), ***Num*** (số lượng) và ***Link*** (chứa địa chỉ nút tiếp theo).

CTDL được khai báo như sau:

```
struct Smartphone
{
  string PhoneName;
  int Num;
  Smartphone *Link;
};
typedef Smartphone* TroPhone;
```

```
struct KhachHang
{
   string Name;
   TroPhone First,Last;
   KhachHang *Next;
};
typedef KhachHang* TroKH;
```

1. Viết hàm **TroKH KTraName(string Name,TroKH Dau)** để kiểm tra khách hàng có tên Name có trong danh sách Dau hay không? Nếu có trả về địa chỉ nút tìm được, ngược lại trả về NULL.
2. Viết hàm **TroPhone KTraPhoneName(string PhoneName,TroPhone First)** để kiểm tra điện thoại có tên PhoneName có trong danh sách First hay không? Nếu có trả về địa chỉ nút tìm được, ngược lại trả về NULL.
3. Viết hàm **void Insert(string PhoneName,string Name,TroKH &Dau)** để bổ sung khách hàng có tên Name sử dụng điện thoại có nhãn hiệu là PhoneName vào danh sách Dau.
4. Viết hàm **int Dem(TroKH Dau)** để đếm xem có bao nhiêu khách hàng sử dụng từ hai loại điện thoại trở lên.

Bài 11: XỬ LÝ VĂN BẢN

Người ta tổ chức lưu trữ các dòng của một văn bản trong bộ nhớ dưới dạng một danh sách liên kết kép mà mỗi nút của nó tương ứng với một dòng. Ta dùng 2 con trỏ *First* và *Last* lần lượt trỏ vào dòng đầu tiên và cuối cùng của văn bản.

CTDL được khai báo như sau:

```
struct TextLine
{
     string Line;
     TextLine *Prev, *Next;
};
typedef TextLine* Text;
```

Giả sử văn bản khác rỗng.

1. Viết hàm **void PrevInsert(string st, Text pos, Text &First)** cho phép chèn một dòng có nội dung là *st* vào trước dòng có địa chỉ là *pos* trong văn bản có nút đầu được trỏ bởi con trỏ *First*.

2. Viết hàm **void NextInsert(string st, Text pos, Text &Last)** cho phép chèn một dòng có nội dung là *st* vào sau dòng có địa chỉ là *pos* trong văn bản có nút cuối được trỏ bởi con trỏ *Last*.
3. Viết hàm **void View(Text First)** để hiển thị văn bản được trỏ bởi First ra màn hình.
4. Viết hàm **void DelLine(Text pos, Text &First, Text &Last)** cho phép xóa dòng tại địa chỉ *pos* trong danh sách được trỏ bởi *First và Last*.
5. Ta định nghĩa *khối* là một dãy liên tiếp các dòng trong văn bản. Ký hiệu (fb,lb) là khối được xác định vị trí bởi địa chỉ dòng đầu *fb* và dòng cuối *lb*. Viết hàm **void CopyBlock(Text fb, Text lb,Text dest, Text &First, Text &Last)** cho phép sao chép khối (fb,lb) tới trước dòng được trỏ bởi *dest* trong danh sách được trỏ bởi *First và Last*. Giả sử *dest* không nằm ở trong khối (fb,lb).
6. Viết hàm **void RemoveBlock(Text fb, Text lb, Text &First, Text &Last)** để xóa khối (fb,lb) trong danh sách được trỏ bởi *First và Last*.
7. Viết hàm **void MoveBlock(Text fb, Text lb,Text dest, Text &First, Text &Last)** để di chuyển khối (fb,lb) tới trước dòng được trỏ bởi *dest* trong danh sách được trỏ bởi *First và Last*. Giả sử *dest* không nằm ở trong khối (fb,lb).

CHƯƠNG 4: DANH SÁCH HẠN CHẾ

4.1. ĐẶT VẤN ĐỀ

Trong các thao tác trên danh sách không phải lúc nào cũng có thể thực hiện được tất cả mà nhiều khi các thao tác này bị hạn chế trong một số loại danh sách mà người ta gọi đó là danh sách hạn chế.

Danh sách hạn chế là danh sách mà các thao tác trên đó bị hạn chế trong một chừng mực nào đó tùy thuộc vào danh sách.

Trong phần này chúng ta xem xét hai loại danh sách hạn chế chủ yếu đó là Ngăn xếp (Stack) và Hàng đợi (Queue). Các danh sách hạn chế này có thể được tổ chức theo danh sách đặc hoặc danh sách liên kết.

4.2. NGĂN XẾP

4.2.1. Định nghĩa ngăn xếp

Ngăn xếp (Stack) là một danh sách mà trong đó thao tác thêm một phần tử vào trong danh sách và thao tác lấy một phần tử ra khỏi danh sách đều được thực hiện ở cùng một đầu.

Ta nói ngăn xếp là một cấu trúc có tính chất ”vào trước – ra sau” hay còn gọi là danh sách LIFO (Last In First Out) hoặc PushDown List.

Ngăn xếp (stack) có thể được tổ chức theo danh sách đặc hoặc danh sách liên kết. Do ở đây cả hai thao tác thêm vào và lấy ra đều được thực hiện ở một đầu nên chúng ta chỉ cần quản lý vị trí đầu của danh sách dùng làm mặt cho ngăn xếp thông qua biến chỉ số bề mặt sp(stack pointer). Chỉ số này có thể là cùng chiều(đầu) hoặc ngược chiều(cuối) với thứ tự các phần tử trong mảng và trong danh sách liên kết. Điều này có nghĩa là bề mặt ngăn xếp có thể là đầu mảng, đầu danh sách liên kết mà cũng có thể là cuối mảng, cuối danh sách liên kết.

4.2.2. Các thao tác trên ngăn xếp

4.2.2.1. Khởi tạo ngăn xếp

Nếu tổ chức ngăn xếp theo danh sách đặc thì khi khởi tạo, ngăn xếp rỗng, ta cho biến sp =0. Nếu tổ chức ngăn xếp theo danh sách liên kết đơn thì khi khởi tạo biến con trỏ sp trỏ vào giá trị NULL.

4.2.2.2. Bổ sung một phần tử vào ngăn xếp

Nếu tổ chức ngăn xếp theo danh sách đặc khi thêm một phần tử vào ngăn xếp ta xét các trường hợp sau: Nếu ngăn xếp bị đầy thì trả về giá trị -1, và trả về giá trị là vị trí của phần tử mới thêm vào ngăn xếp.

4.2.2.3. Lấy một phần tử ra khỏi ngăn xếp

Nếu tổ chức ngăn xếp theo danh sách đặc khi loại bỏ một phần tử ra khỏi ngăn xếp ta xét các trường hợp sau: Nếu ngăn xếp rỗng thì thông báo ngăn xếp rỗng không thực hiện loại bỏ, ngược lại ngăn xếp không rỗng thì lấy giá trị của ngăn xếp ra, và giá trị của sp=sp-1.

Nếu tổ chức ngăn xếp theo danh sách liên kết đơn thì khi loại một phần tử ra khỏi ngăn xếp thì trả về giá trị là địa chỉ của phần tử phía dưới nó(tức là tiếp theo nó) và xóa giá trị ngay tại vị trí cần loại bỏ.

4.2.3. Cài đặt ngăn xếp

4.2.3.1. Cài đặt ngăn xếp bằng mảng

- **Khởi tạo ngăn xếp**

```
#define MAX 20
int Stack[MAX];
int Top = 0;
```

- **Thêm một phần tử vào ngăn xếp (PUSH)**

```
void PUSH(int x,int &Top,int Stack[])
{
  Top++;
  Stack[Top] = x;
}
```

- **Lấy một phần tử ra khỏi ngăn xếp (POP)**

```
int POP(int &Top,int Stack[])
{
  int x = Stack[Top];
  Top--;
  return x;
}
```

4.2.3.2. Cài đặt ngăn xếp bằng danh sách liên kết

- **Khởi tạo ngăn xếp**

```
struct NUT
{
  int Data;
  NUT *next;
```

```
};
typedef NUT* Tro;
```

➢ **Thêm một phần tử vào ngăn xếp (PUSH)**

```
void PUSH(int x,Tro &Stack)
{
  Tro p=new(NUT);
  p->Data = x;
  p->next = Stack;
  Stack = p;
}
```

➢ **Lấy một phần tử ra khỏi ngăn xếp (POP)**

```
int POP(Tro &Stack)
{
  Tro p = Stack;
  Stack = Stack->next;
  int x = p->Data;
  delete(p);
  return x;
}
```

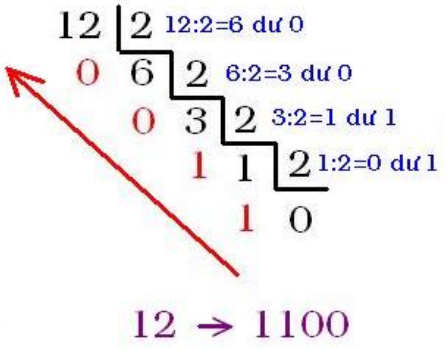
4.2.4. Ứng dụng của ngăn xếp

4.2.4.1. Khử đệ quy

Một chương trình có sử dụng giải thuật đệ quy thì sẽ ngắn gọn, dễ cài đặt, tuy nhiên sự hạn chế của bộ nhớ dành cho chương trình không cho phép chúng ta xây dựng chương trình bằng đệ quy. Vì vậy, việc chuyển tchương trình từ dạng đệ quy sang không đệ quy (chỉ sử dụng các cấu trúc lặp) gọi là khử đệ quy.

Khử đệ quy giúp chúng ta vẫn giữ được nguyên bản thuật toán đệ quy của mình nhưng trong chương trình không hề có lời gọi đệ quy, và như thế chương trình có thể chạy được trong bất kỳ môi trường lập trình nào. Stack là một trong những công cụ hữu hiệu cho việc khử đệ qui.

Ví dụ: Viết giải thuật để biểu diễn nhị phân của một số nguyên dương n.



Ta có thể thực hiện theo các bước sau:

Bước 1: Khai báo stack S rỗng

Bước 2: Trong khi n>0 thì tiến hành

```
{
        PUSH(x%2,S);
        n = n/2;
}
```

Bước 3: Lấy các giá trị từ ngăn xếp ra và in ra màn hình.

```
void Bin(int n)
{
  Tro Stack=NULL;
  while(n>0)
  {
    PUSH(n%2,Stack);
    n=n/2;
  }
  while(Stack)
  {
      int x = POP(Stack);
      cout<<x;
  }
}
```

4.2.4.2. Định giá biểu thức toán học

Giải thuật định giá biểu thức toán học gồm hai công đoạn sau:

Giai đoạn 1: Chuyển biểu thức toán học từ dạng trung tố sang dạng hậu tố.

Giai đoạn 2: Tính giá trị biểu thức hậu tố vừa tìm được ở giai đoạn 1.

<u>Giai đoạn 1</u>: Áp dụng ký pháp nghịch đảo Balan để chuyển biểu thức toán học từ dạng trung tố sang dạng hậu tố:

Một biểu thức toán học thông thường có thể được biến đổi thành dãy gồm các toán tử và toán hạng, trong đó một toán tử được dùng để tính toán với 1 hoặc 2 toán hạng đứng trước nó trong dãy đó.

Ví dụ:

(A+B)*C biến đổi thành AB+C*
A+B*C biến đổi thành ABC*+

Giải thuật: Chuyển biểu thức **Infix** ở dạng trung tố sang dạng hậu tố **Posfix** như sau:

***Bước 1:**

- Khởi gán: **Posfix** = " ";
- Khởi tạo ngăn xếp S = ∅;

***Bước 2:** Xét từng phần tử X thuộc **Infix** (từ trái sang phải):

- Nếu X là dấu "(": `PUSH(X,S);`
- Nếu X là dấu ")":

```
Token = POP(Stack);
while(Token!="(")
{
    Posfix = Posfix + Token;
    Token = POP(Stack);
}
```

✍***Chú ý**: dấu mở ngoặc "(" cũng lấy ra khỏi S nhưng không cho vào* **Posfix**.

- Nếu X là toán hạng: **`Posfix = Posfix + X;`**
- Nếu X là toán tử:

+ ***while***(toán tử ở đỉnh Stack ≥ X) **`Posfix = Posfix + POP(S);`**

+ `PUSH(X,S);`

***Bước 3:** Lấy các phần tử còn trong stack S bổ sung vào cuối **Posfix**.

Ví dụ: Chuyển biểu thức **A*B+C*((D-E)+F)/G** từ dạng trung tố sang dạng hậu tố:

Infix	Token	Stack	Posfix
A*B+C*((D-E)+F)/G	A	{Empty}	A
***B+C*((D-E)+F)/G**	*	*	A
B+C*((D-E)+F)/G	B	*	A B
+C*((D-E)+F)/G	+	+	A B *
C*((D-E)+F)/G	C	+	A B * C
***((D-E)+F)/G**	*	+ *	A B * C
((D-E)+F)/G	(	+ * (	A B * C
(D-E)+F)/G	(	+ * ((	A B * C
D-E)+F)/G	D	+ * ((	A B * C D
-E)+F)/G	-	+ * ((-	A B * C D
E)+F)/G	E	+ * ((-	A B * C D E
)+F)/G	)	+ * (	A B * C D E -
+F)/G	+	+ * (+	A B * C D E -

F)/G	F	+ * (+	A B * C D E – F
)/G	)	+ *	A B * C D E – F +
/G	/	+ /	A B * C D E – F + *
G	G	+ /	A B * C D E – F + * G
		{Empty}	A B * C D E – F + * G / +

Giai đoạn 2: Tính giá trị biểu thức hậu tố.

***Bước 1:** Khởi tạo stack S = ∅;

***Bước 2:** Xét từng phần tử X của **Posfix**:

- Nếu X là toán hạng: PUSH(X,S);
- Nếu X là toán tử:
  + b = POP(S);
  + a = POP(S);
  + Tính z = a **X** b với X là toán tử của **Posfix**.
  + PUSH(z,S);

***Bước 3:** return POP(S);

Ví dụ: Tính giá trị biểu thức hậu tố 32+64-*

Posfix	**Stack**
32+64-*	∅
2+64-*	3
+64-*	3 2
64-*	5
4-*	5 6
-*	5 6 4
*	5 2
” ”	**10**

4.3. HÀNG ĐỢI

4.3.1. Định nghĩa

Hàng đợi là một danh sách mà trong đó thao tác thêm một phần tử vào trong danh sách được thực hiện ở một đầu và thao tác lấy một phần tử từ trong danh sách ra lại được thực hiện ở đầu kia.

Như vậy hàng đợi hoạt động theo kiểu FIFO (First In First Out – Vào trước ra trước).

4.3.2. Các phép toán trên hàng đợi

4.3.2.1. Khởi tạo hàng đợi

Khi khởi tạo hàng đợi thì hàng đợi rỗng, chưa có phần tử nào. Ta có thể tạo hàm Creat_Queue(Q) để khởi tạo hàng đợi.

4.3.2.2. Bổ sung một phần tử vào hàng đợi

Khi thêm một phần tử vào hàng đợi thì phải kiểm tra hàng đợi đã đầy hay chưa, nếu chưa đầy thì ta thêm phần tử đó vào cuối hàng đợi ngược lại thì thông báo lỗi tràn hàng đợi.

4.3.2.3. Lấy một phần tử ra khỏi hàng đợi

Khi muốn lấy một phần tử ra khỏi hàng đợi thì phải kiểm tra hàng đợi có rỗng không, nếu không rỗng thì mới thực hiện được.

4.3.3. Cài đặt hàng đợi

4.3.3.1. Cài đặt hàng đợi bằng mảng

Ta có thể tạo một hàng đợi bằng cách sử dụng mảng 1 chiều với kích thước tối đa là N (ví dụ, N có thể bằng 1000) theo kiểu xoay vòng (coi phần tử a_{n-1} kề với phần tử a_0).

- **Khởi tạo hàng đợi**

```
int front = rear = 0;
```

- **Kiểm tra xem hàng đợi có rỗng không**

Để kiểm tra hàng đợi có rỗng hay không ta chỉ cần kiểm tra giá trị **rear** của hàng đợi. Nếu **rear =0** chứng tỏ hàng đợi này rỗng.

- **Kiểm tra hàng đợi đã đầy chưa**

Để kiểm tra hàng đợi đã đầy hay chưa, ta chỉ cần kiểm tra giá trị rear – front của hàng đợi. Nếu **rear – front = N** (là số phần tử cực đại của mảng) thì hàng đợi bị đầy.

- **Thêm một phần tử vào cuối hàng đợi**

Khi thêm một phần tử vào hàng đợi ta xem xét các trường hợp sau:

- Nếu hàng đợi rỗng: thêm giá trị X vào phần tử có chỉ số rear, tăng rear lên một đơn vị.
- Ngược lại: kiểm tra xem hàng đợi có bị tràn hay không, nếu bị tràn nhưng dữ liệu chưa đầy thì tiến hành giải quyết hàng tràn theo một trong hai cách:

Cách 1: Tịnh tiến hàng lên **front - 1** phần tử.

Cách 2: Xem như hàng là vòng tròn, thêm dữ liệu vào phần tử đầu tiên của mảng trước front, nếu **rear = N** thì **rear = 0** ngược lại tăng **rear** lên 1 đơn vị.

4.3.3.2. Cài đặt hàng đợi bằng danh sách liên kết

- **Khởi tạo hàng đợi**

Cho hai biến con trỏ First và Last (trỏ tới phần tử đầu và cuối danh sách) bằng NULL.

- **Kiểm tra xem hàng đợi có rỗng không**

  Hàng đợi rỗng khi First = NULL.
- **Thêm một phần tử vào hàng đợi**

  Khi thêm một phần tử mới vào hàng đợi, ta thêm nó vào cuối hàng đợi.
- **Lấy ra một phần tử từ hàng đợi**

  Khi lấy một phần tử ra khỏi hàng đợi, ta lấy phần tử đầu tiên của hàng đợi.

BÀI TẬP

Bài 1: Sử dụng cơ chế stack để chuyển các biểu thức sau sang dạng hậu tố:

a. P1= (a+b)*(c-d)
b. P2= (a*n+(b-c)*k)/h

Bài 2:

Viết chương trình thực hiện các công việc sau:

- Chuyển biểu thức từ trung tố sang hậu tố.
- Tính giá trị biểu thức hậu tố.

Bài 3: Ứng dụng ngăn xếp viết chương trình cho phép nhập vào một xâu và in xâu đảo ngược của xâu đó ra màn hình.

Bài 4: Hoàn thiện chương trình cài đặt ngăn xếp bằng mảng và bằng danh sách liên kết với các khai báo và các thao tác thực hiện được trên ngăn xếp đã học.

Bài 5: Hoàn thiện chương trình cài đặt hàng đợi bằng mảng và bằng danh sách liên kết với các khai báo và các thao tác thực hiện được trên ngăn xếp đã học.

CHƯƠNG 5: CÂY

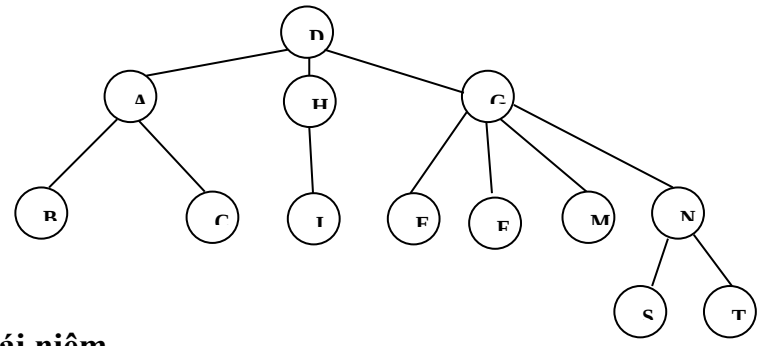
5.1. ĐỊNH NGHĨA VÀ MỘT SỐ KHÁI NIỆM

5.1.1. Định nghĩa

Cây là một cấu trúc dữ liệu gồm một tập hữu hạn các nút, giữa các nút có một quan hệ phân cấp gọi là quan hệ "cha-con". Có một nút đặc biệt gọi là nút gốc (root).

5.1.2. Biểu diễn cây

Để biểu diễn cây có rất nhiều cách, dùng đồ thị là một cách biểu diễn cơ bản nhất:



5.1.3. Các khái niệm

Nút gốc

Nút gốc là nút duy nhất không có nút cha.

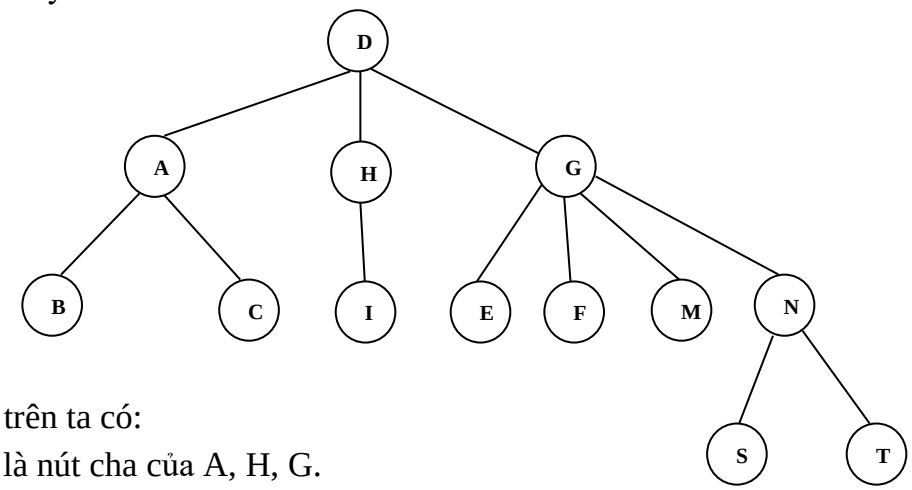
Nút trước và nút sau

Nút T được gọi là nút trước 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 của T.

Cấp của nút

Số các con của một nút được gọi là cấp của nút đó.

Ví dụ: Cho cây như hình sau:

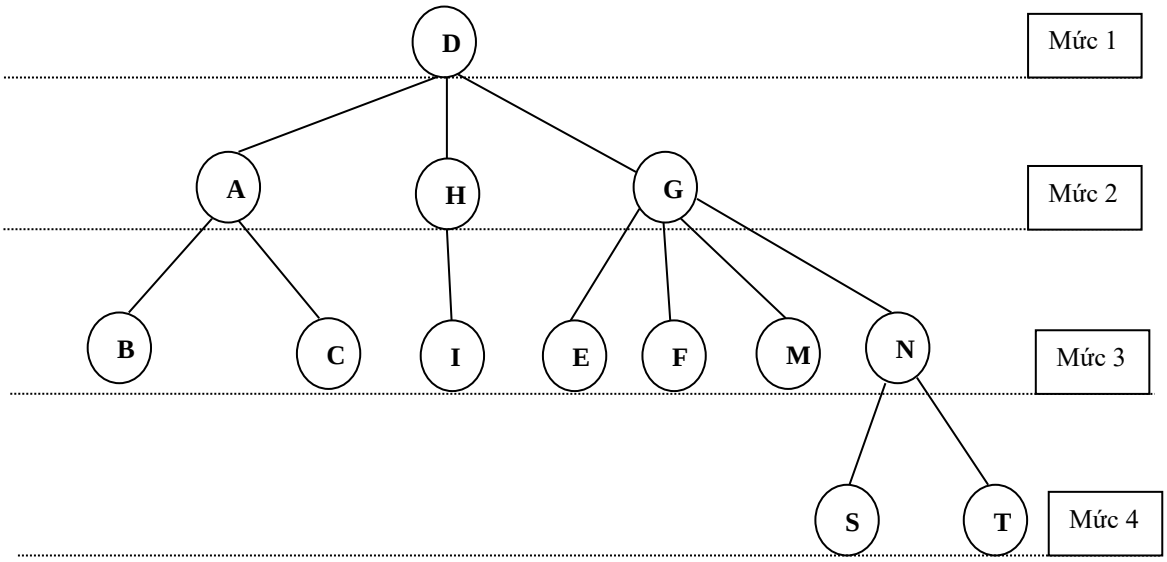


Trong cây trên ta có:

- D là nút cha của A, H, G.
- B, C là nút con của nút A.
- Nút D được gọi là nút gốc(root).

Mức của cây

Gốc của cây người ta gán cho mức 1, nếu nút cha có mức là i thì nút con sẽ có mức là i+1. Mức của một nút bằng mức của node gốc cây con chứa nó cộng thêm 1.



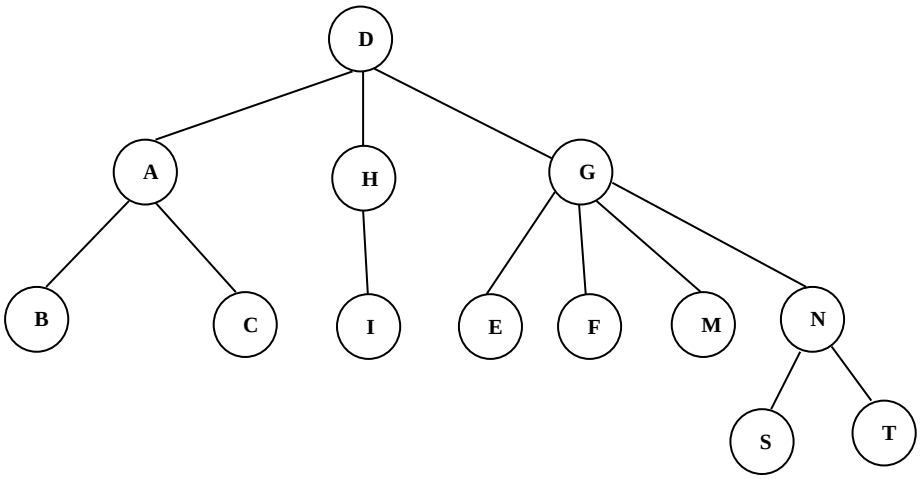
Nút cha và nút con

Nút P được gọi là nút cha của nút C nếu nút P là nút trước của nút C và mức của nút C lớn hơn mức của nút P là 1 mức. Khi đó nút C được gọi là nút con của nút P.

Nút lá và nút nhánh

Nút lá: Nút có cấp bằng 0 được gọi là nút lá (leaf) hay là nút tận cùng.

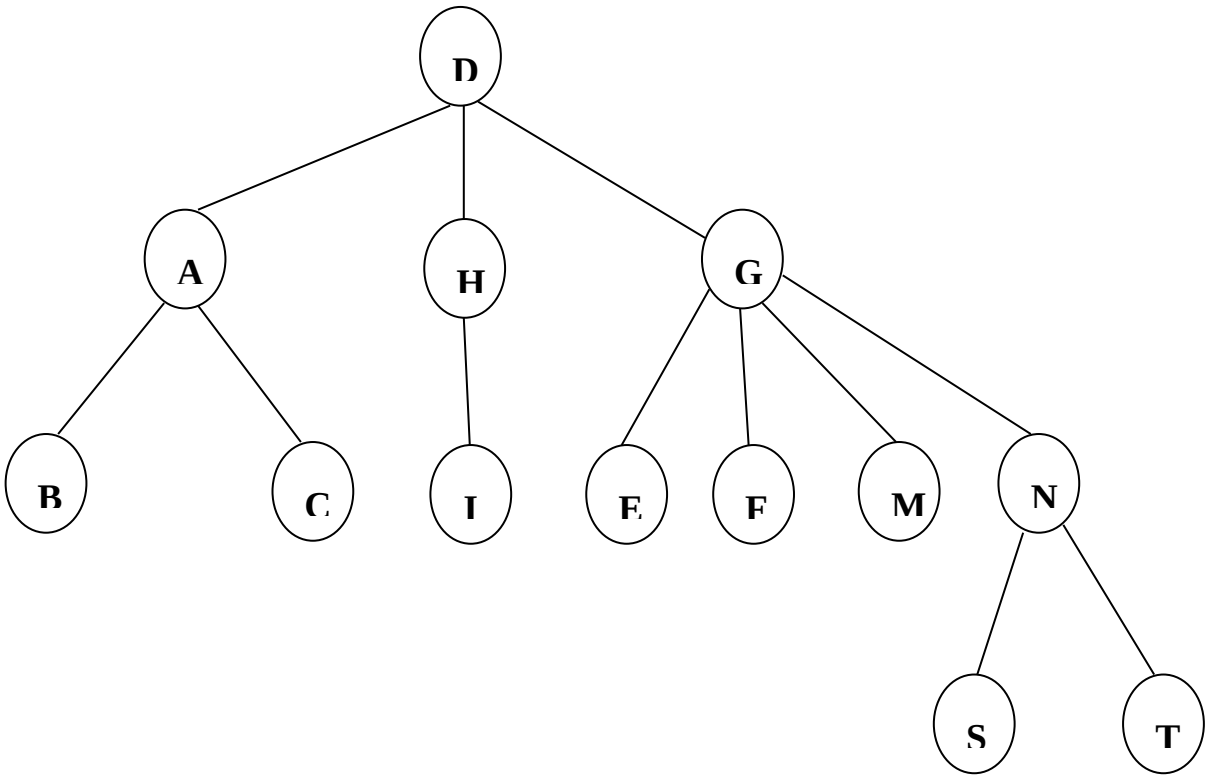
Nút nhánh: Những nút không phải là nút lá (có bậc khác 0) và không phải là nút gốc thì được gọi là nút nhánh (branch).



Trong cây trên thì các nút B, C, I, E, F, M, S,T được gọi là các nút lá(leaf) hay còn gọi là nút tận cùng.

Cấp của cây

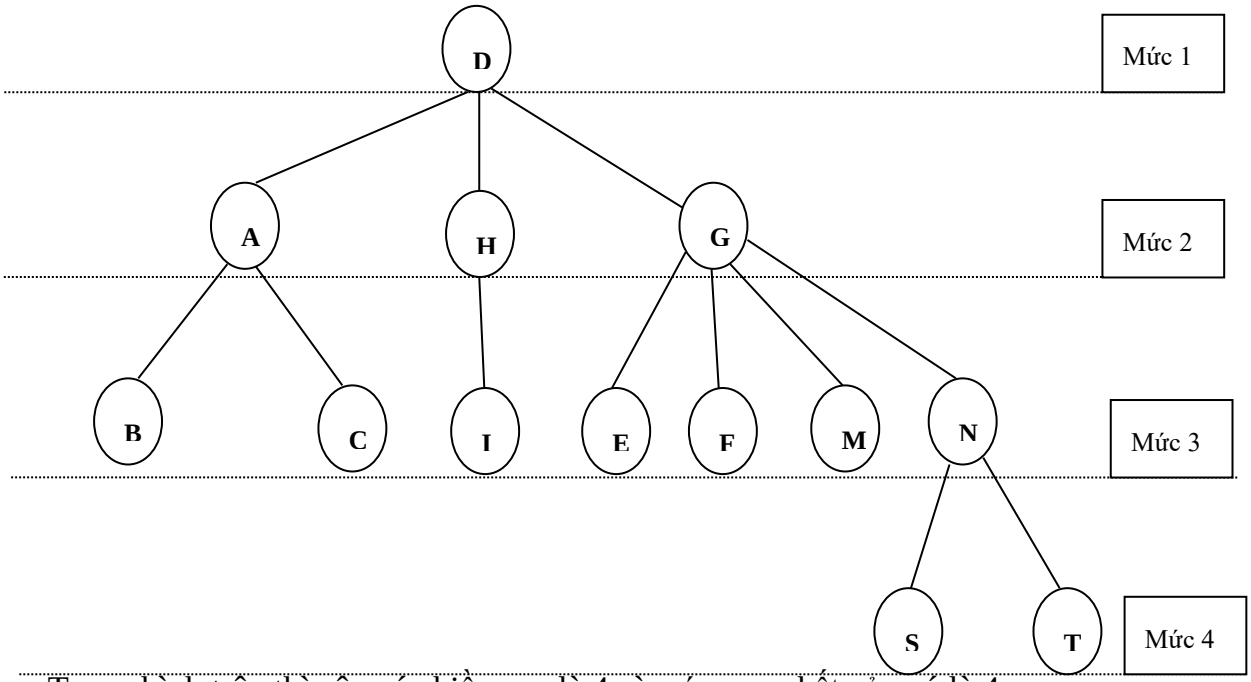
Cấp cao nhất của một nút trên cây được gọi là cấp của cây đó.



Trong cây trên thì cấp của cây là cấp 4 vì nút G có cấp 4 là cấp cao nhất của cây.

Chiều cao (chiều sâu) của cây

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



Trong hình trên thì cây có chiều cao là 4 vì mức cao nhất của nó là 4.

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. Như vậy, 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 của nó cộng thêm 1. Chiều dài của một nút tại mức thứ i là i.

Chiều dài đường đi của cây

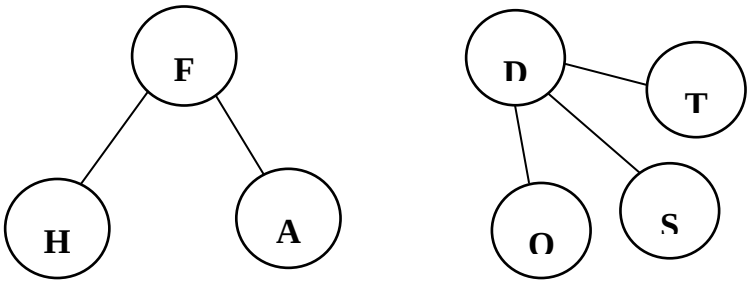
Chiều dài đường đi của một cây là tổng tất cả các chiều dài đường đi của tất cả các nút trên cây. Đây chính là chiều dài đường đi trong của cây. Đường đi trung bình của cây chính là chiều dài của cây chia cho tổng số nút của cây.

Rừng cây (Forest)

Rừng là tập hợp các cây.

Như vậy một cây khi loại bỏ nút gốc sẽ cho ta một rừng cây.

Ví dụ: Ta có rừng cây như sau



5.1.4. Một số tính chất của cây

- Nút gốc không có nút cha.
- Các nút khác chỉ có một nút cha.
- Mỗi nút có thể có nhiều nút con.
- Cây không có chu trình.

5.2. CÂY NHỊ PHÂN

5.2.1. Định nghĩa và các khái niệm bổ sung

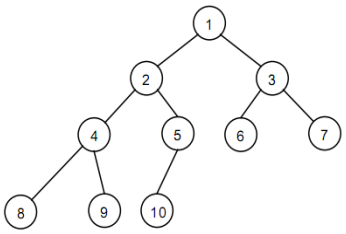
Định nghĩa cây nhị phân

Cây nhị phân là cây mà mọi nút trên cây chỉ có tối đa hai nhánh con. Với mỗi nút người ta có cây con trái và cây con phải của nút đó.

Các khái niệm bổ sung

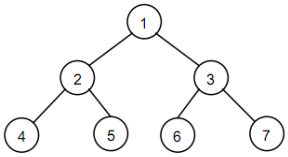
Cây nhị phân hoàn chỉnh

Là cây nhị phân mà mọi nút tại mức nhỏ hơn mức (h -1) đều có đúng hai nút con -với h là chiều cao của cây đó.

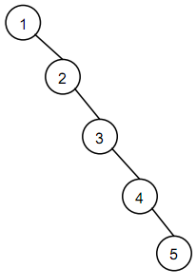


Cây nhị phân đầy đủ

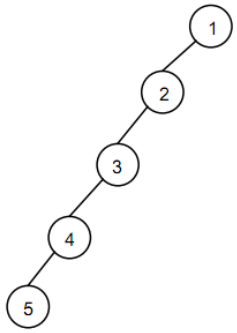
Là cây nhị phân mà mọi nút tại mức nhỏ hơn **hay bằng** mức (h -1) đều có đúng hai nút con -với h là chiều cao của cây đó.



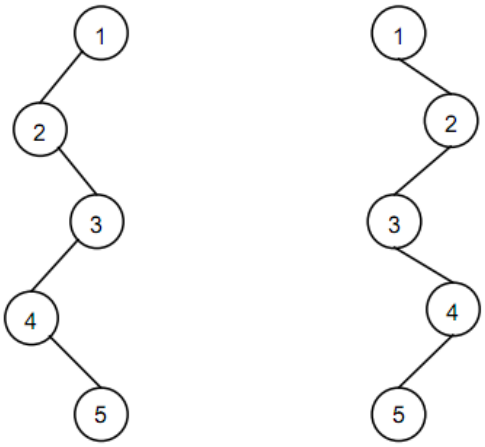
Cây lệch phải là cây nhị phân chỉ có cây con phải (Các nút trên cây chỉ có con phải mà không có con trái.)



Cây lệch trái là cây nhị phân chỉ có cây con trái (Các nút trên cây chỉ có con trái không có con phải).



Cây zíc zắc là cây nhị phân mà node trái và node phải của cây đan xen nhau thành một hình zíc zắc.



Các loại cây lệch phải, cây lệch trái, cây zíc zắc được gọi là cây nhị phân suy biến.

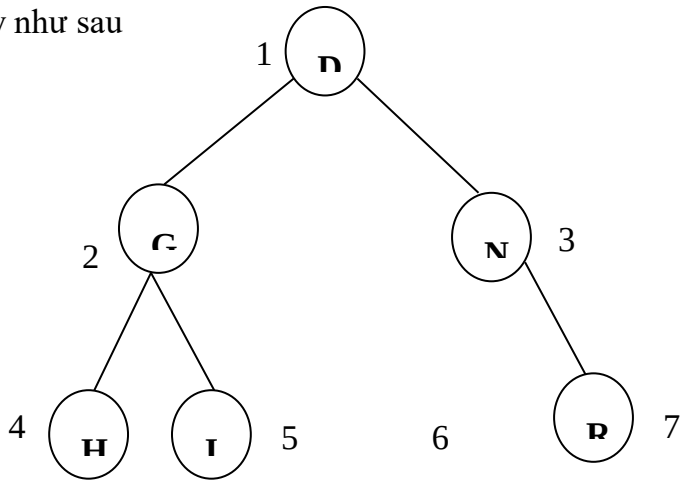
5.2.2. Tổ chức lưu trữ cây nhị phân trong máy tính

Để lưu trữ cây nhị phân trong máy tính ta có thể lưu trữ bằng mảng hoặc bằng danh sách liên kết.

5.2.2.1. Lưu trữ cây nhị phân bằng mảng

Nguyên tắc: Đánh số trên cây nhị phân theo thứ tự từ trên xuống và từ trái sang phải(tính luôn cả những node bị khuyết để tạo thành một cây nhị phân hoàn chỉnh). Chỉ số được đánh chính là chỉ số của phần tử mảng chứa node đó.

Ví dụ: Cho cây như sau



D	G	N	H	I		R
1	2	3	4	5	6	7

Theo cách lưu trữ trên mảng thì ta thấy:

- Nếu node cha ở vị trí thứ i thì node con trái của nó có vị trí là 2*i, node con phải của nó có vị trí là 2*i+1.

- Nếu node con trái có vị trí thứ i, thì node cha của nó có vị trí là i/2.

Việc lưu trữ dữ liệu của cây nhị phân trên mảng chỉ thích hợp với cây nhị phân đầy đủ hay cây nhị phân hoàn chỉnh. Còn đối với các dạng khác thì việc lưu trữ bằng mảng sẽ gây ra hiện tượng lãng phí bộ nhớ.

5.2.2.2. Lưu trữ cây nhị phân bằng danh sách liên kết

Ngoài việc lưu trữ cây nhị phân bằng mảng người ta còn có thể lưu trữ cây nhị phân bằng danh sách liên kết kép. Với mỗi node ta dùng hai vùng liên kết Left và Right, vùng liên kết Left chỉ đến node con bên trái, vùng liên kết Right chỉ đến node con bên phải và một trường dữ liệu Data chứa nội dung của node đang xét.

Cấu trúc một node được mô tả như hình sau:

Left	Data	Right

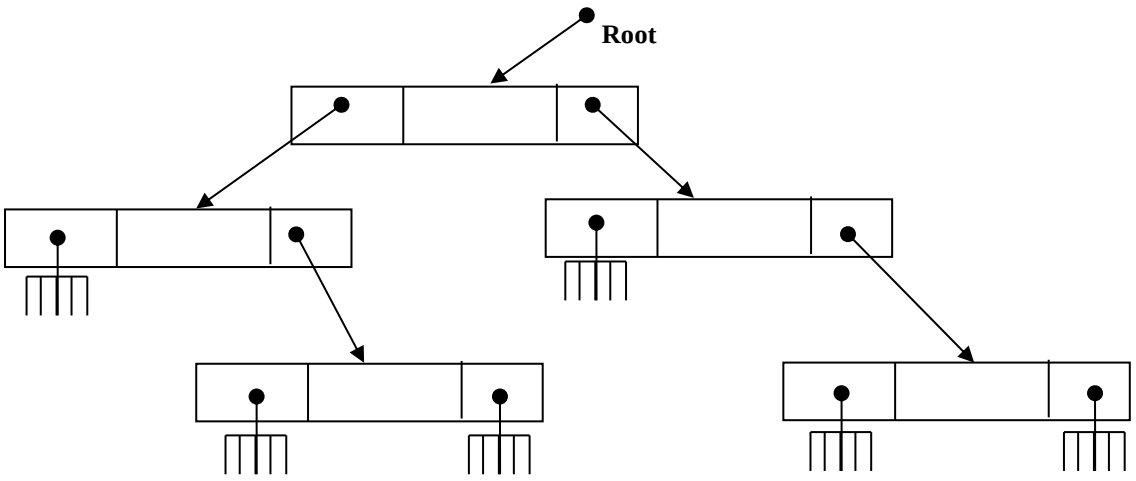
Ta có thể khai báo một node của cây nhị phân như sau:

```
struct NODE
{
    <Type> Data;
    NODE *Left, *Right;
};
typedef NODE* TREE;
```

Khi đó, ta có thể khai báo một biến con trỏ Root trỏ vào node gốc của cây nhị phân:

```
TREE Root;
```

Ta có thể hình dung một cây được lưu trữ dưới dạng danh sách liên kết như hình sau:



5.2.3. Các phép duyệt trên cây nhị phân

5.2.3.1. Duyệt theo thứ tự gốc trước (NLR)

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

- Duyệt nút gốc trước, duyệt cây con trái, duyệt cây con phải(Root-Left-Right).
- Duyệt nút gốc trước, duyệt cây con phải, duyệt cây con trái(Root-Right-Left).

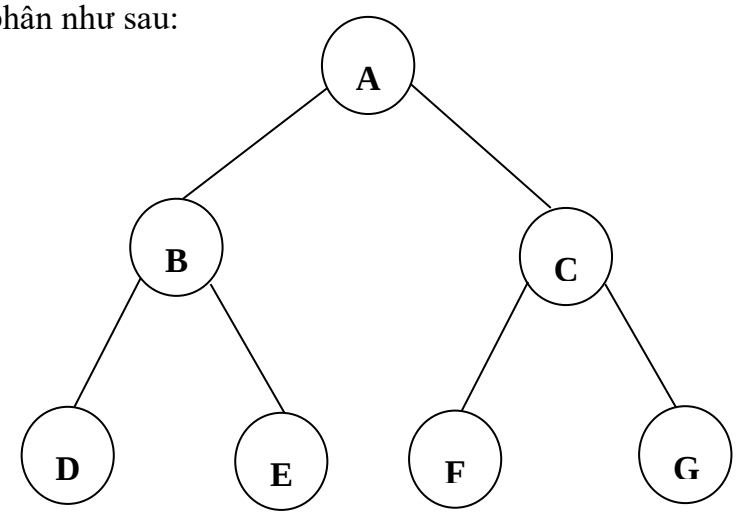
Giải thuật duyệt cây theo thứ tự gốc trước:

Bước 1: Nếu cây rỗng thì không làm gì.

Bước 2: Ngược lại nếu cây không rỗng thì thực hiện:

- Thăm node gốc của cây
- Duyệt cây con bên trái (hoặc bên phải) theo thứ tự trước
- Duyệt cây con bên phải (hoặc bên trái) theo thứ tự trước

Ví dụ: Cho cây nhị phân như sau:



Kết quả duyệt theo Root-Left-Right như sau:

A→B→D→E→C→F→G

Kết quả duyệt theo Root-Right-Left như sau:

A→C→G→F→B→E→D

5.2.3.2. Duyệt theo thứ tự nút gốc giữa (LNR)

Theo cách duyệt này thì chúng ta duyệt một trong hai cây con trước rồi mới duyệt nút gốc sau đó mới duyệt cây con còn lại. Do đó căn cứ vào thứ tự duyệt hai cây con mà chúng ta có hai cách duyệt theo thứ tự nút gốc giữa như sau:

- 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).

Giải thuật duyệt cây theo thứ tự nút gốc giữa

Bước 1: Nếu cây rỗng thì không làm gì.

Bước 2: Nếu cây không rỗng thì thực hiện:

- Duyệt cây con bên trái (hoặc bên phải) theo thứ tự giữa.
- Thăm node gốc của cây.
- Duyệt cây con bên phải (hoặc bên trái) theo thứ tự giữa.

Kết quả duyệt theo Left-Root-Right như sau:

D→B→E→A→F→C→G

Kết quả duyệt theo Right-Root-Left như sau:

G→C→F→A→E→B→D

5.2.3.3. Duyệt theo thứ tự nút gốc sau (LRN)

Theo cách duyệt này thì chúng ta duyệt hai cây con trước rồi mới duyệt nút gốc . Do đó căn cứ vào thứ tự duyệt hai cây con mà chúng ta có hai cách duyệt theo thứ tự nút gốc sau như 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).

Giải thuật duyệt cây theo thứ tự nút gốc sau

Bước 1: Nếu cây rỗng thì không làm gì.

Bước 2: Nếu cây không rỗng thì thực hiện:

- Duyệt cây con bên trái (hoặc bên phải) theo thứ tự gốc sau.
- Duyệt cây con bên phải (hoặc bên trái) theo thứ tự gốc sau.
- Thăm node gốc của cây.

Kết quả duyệt theo Left-Right-Root như sau:

D→E→B→F→G→C→A

Kết quả duyệt theo Right-Left-Root như sau:

G→F→C→E→D→B→A

5.2.3.3. Duyệt cây theo mức

Ý tưởng của phương pháp này là sử dụng hàng đợi để duyệt.

Bước 1: Khởi tạo hàng đợi Q = NULL;

Bước 2: Đưa nút gốc vào Q;

Bước 3: Trong khi hàng đợi Q khác rỗng:

- Lấy 1 nút p từ hàng đợi, thăm p->Data;
- Đưa 2 nút con của p (nếu có) vào hàng đợi.

Giải thuật trên có thể được cài đặt như sau:

```
void DuyetMuc(Nut Root)
{
    Queue Q=NULL;
    PUSH(Root,Q);
    while(Q)
    {
```

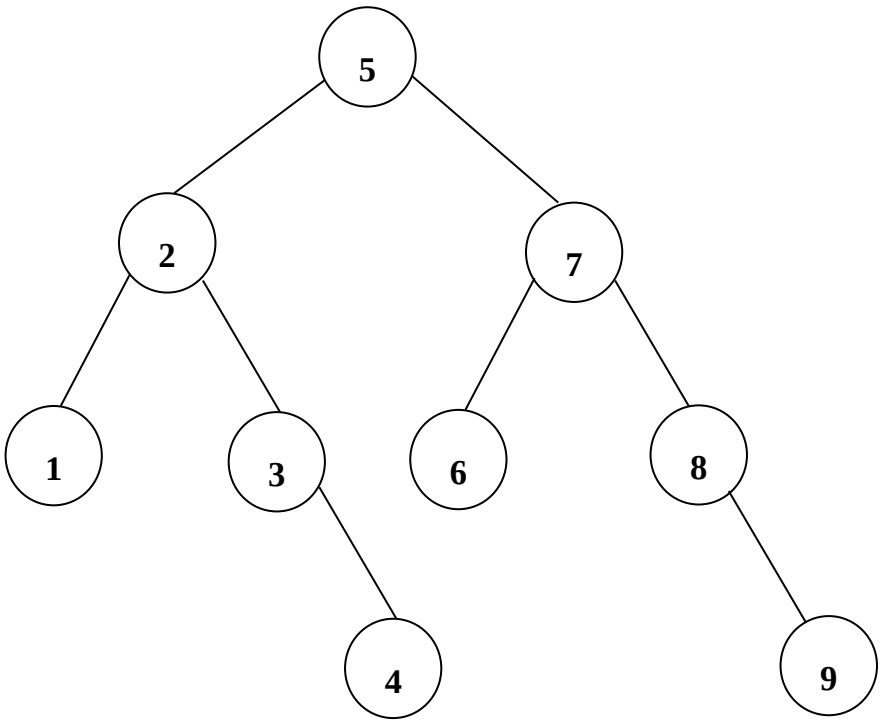
```
            Nut p=POP(Q);
            <Thăm p->Data>;
            if(p->Left) PUSH(p->Left,Q);
            if(p->Right) PUSH(p->Right,Q);
        }
    }
```

5.3. CÂY TÌM KIẾM NHỊ PHÂN

5.3.1. Định nghĩa

Cây tìm kiếm nhị phân (BST – Binary Search Tree) là cây nhị phân trong đó tại mỗi nút, khóa của nút đang xét lớn hơn khóa của tất cả các nút thuộc cây con trái và nhỏ hơn khóa của tất cả các nút thuộc cây con phải.

Cây trong các hình vẽ sau là một cây nhị phân tìm kiếm:



5.3.2. Tính chất cây nhị phân tìm kiếm

Khi duyệt cây theo thứ tự giữa (LNR) ta được một dãy có thứ tự tăng dần.

5.3.3. Các thao tác trên cây BST

5.3.3.1. Khởi tạo cây

```
TREE Root = NULL;
```

5.3.3.2. Duyệt cây

Để duyệt cây ta có 3 cách duyệt nhưng ở phần này chỉ xét đến phần duyệt theo thứ tự LNR để có thể thấy rõ hơn tính chất của cây nhị phân tìm kiếm. Hàm hiển thị nội dung cây theo thứ tự LNR như sau:

```
void LNR(TREE Root)
{
   if (Root)
   {
      LNR(Root->Left);
      <Thăm Root->Data>;
      LNR(Root->Right);
   }
}
```

5.3.3.3. Tìm kiếm một giá trị X trên cây

Muốn tìm kiếm một giá trị X có trên cây nhị phân tìm kiếm hay không ta thực hiện theo các bước như sau:

Giải thuật đệ quy:

Bước 1: Nếu cây rỗng thì trả về giá trị 0: Tức là không có X trên cây

Ngược lại thì chuyển qua bước 2

Bước 2:

Nếu giá trị tại node gốc của cây bằng X thì trả về 1

Nếu X lớn hơn giá trị tại node gốc của cây thì gọi lại lời gọi tìm kiếm X trên cây con phải. Ngược lại thì gọi lại lời gọi tìm kiếm X trên cây con trái.

Giải thuật không đệ quy:

Bước 1: Nếu cây rỗng thì trả về giá trị 0: Tức là không có X trên cây

Ngược lại thì chuyển qua bước 2

Bước 2: Bắt đầu tại node gốc của cây, xét các trường hợp sau:

- Nếu X bằng với giá trị tại node gốc này thì trả về giá trị 1: có X trong cây.
- Nếu X nhỏ hơn giá trị tại node gốc này thì tìm tiếp node con bên trái của node hiện tại, ngược lại tìm tiếp node con bên phải của node hiện tại.

Bước 3:

Nếu node cần xét tiếp theo bằng NULL thì kết luận không có node X trong cây ngược lại cho tìm ở node tiếp theo, lặp lại bước 2 cho đến khi các node trên cây được duyệt hoàn toàn.

5.3.3.4. Thêm một phần tử X vào cây

Để thêm một phần tử X vào cây ta tiến hành theo các bước như sau:

Bước 1: Ta tiến hành tìm kiếm X trong cây, nếu X đã có trong cây thì không thêm X vào nữa, ngược lại X chưa có trong cây thì chuyển qua bước 2:

Bước 2:

- Xét tại node gốc của cây:

- Nếu node gốc bằng NULL thì tạo một node mới với giá trị là X và thêm node vào cây. Nếu X bằng với giá trị tại node gốc thì giải thuật dừng vì không cần thêm node vào cây.

- Nếu X lớn hơn giá trị tại node gốc của cây thì tiến hành gọi đệ quy cho cây con phải, ngược lại thì gọi đệ quy cho cây con trái.

5.3.3.5.Xóa một phần tử có khóa X

Các bước giải thuật hủy một phần tử có khóa X trên cây.

Muốn xóa một node có chứa khóa X ta tiến hành theo các bước sau đây:

Bước 1: Tìm kiếm xem có node nào chứa khóa X trên cây hay không, nếu không có node nào chứa khóa X thì kết thúc giải thuật, ngược lại chuyển qua bước 2.

Bước 2: Xét ba trường hợp sau:

Có 3 trường hợp khi hủy nút X có thể xảy ra:

i. X có đủ cả 2 con.

ii. X là nút lá

iii. X chỉ có 1 con (trái hoặc phải).

Trường hợp 1: Thay vì xóa X, ta sẽ tìm một phần tử thế mạng Y. Phần tử này có tối đa một con. Thông tin lưu tại Y sẽ được chuyển lên lưu tại X.

Vấn đề là phải chọn Y sao cho khi lưu Y vào vị trí của X, cây vẫn là cây BST.

Có 2 phần tử thỏa mãn yêu cầu:

- Phần tử nhỏ nhất (trái nhất) trên cây con phải.
- Phần tử lớn nhất (phải nhất) trên cây con trái.

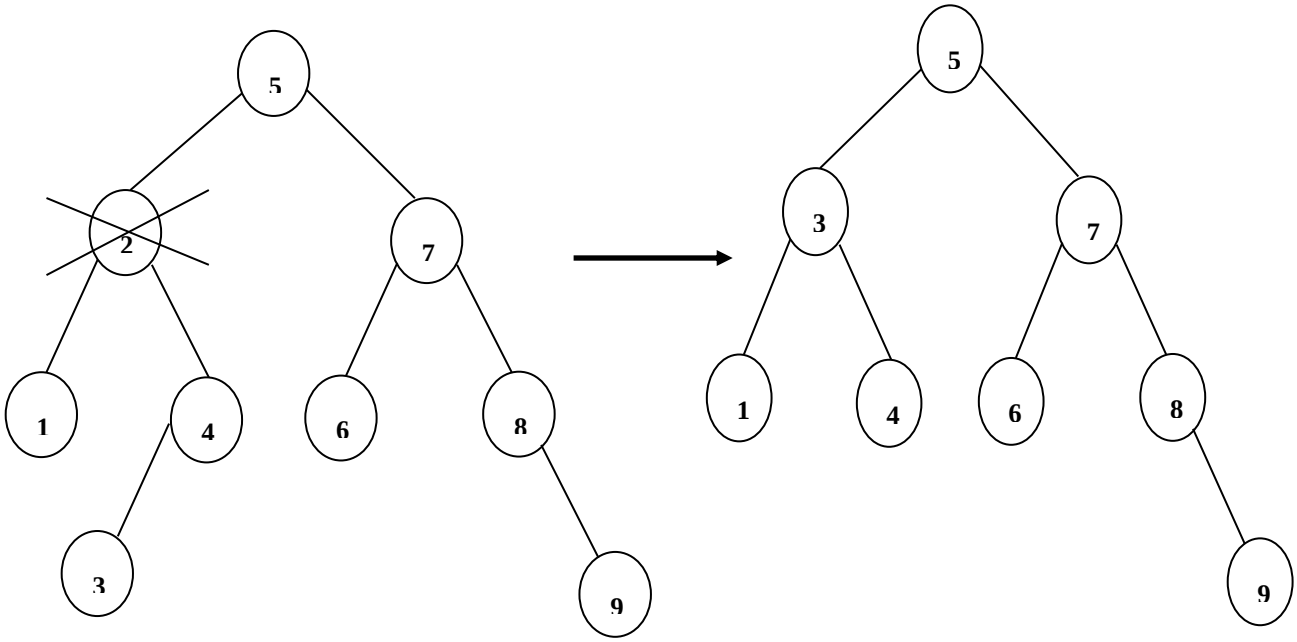
Việc chọn lựa phần tử nào là phần tử thế mạng hoàn toàn phụ thuộc vào ý thích của người lập trình (***Giả sử chọn phần tử trái nhất trên cây con phải làm phân tử thế mạng***).

Trường hợp 2: Nếu node chứa khóa X là node lá thì ta cho con trỏ của node cha của node chứa khóa X trỏ vào NULL, rồi tiến hành hành xóa node chứa khóa X.

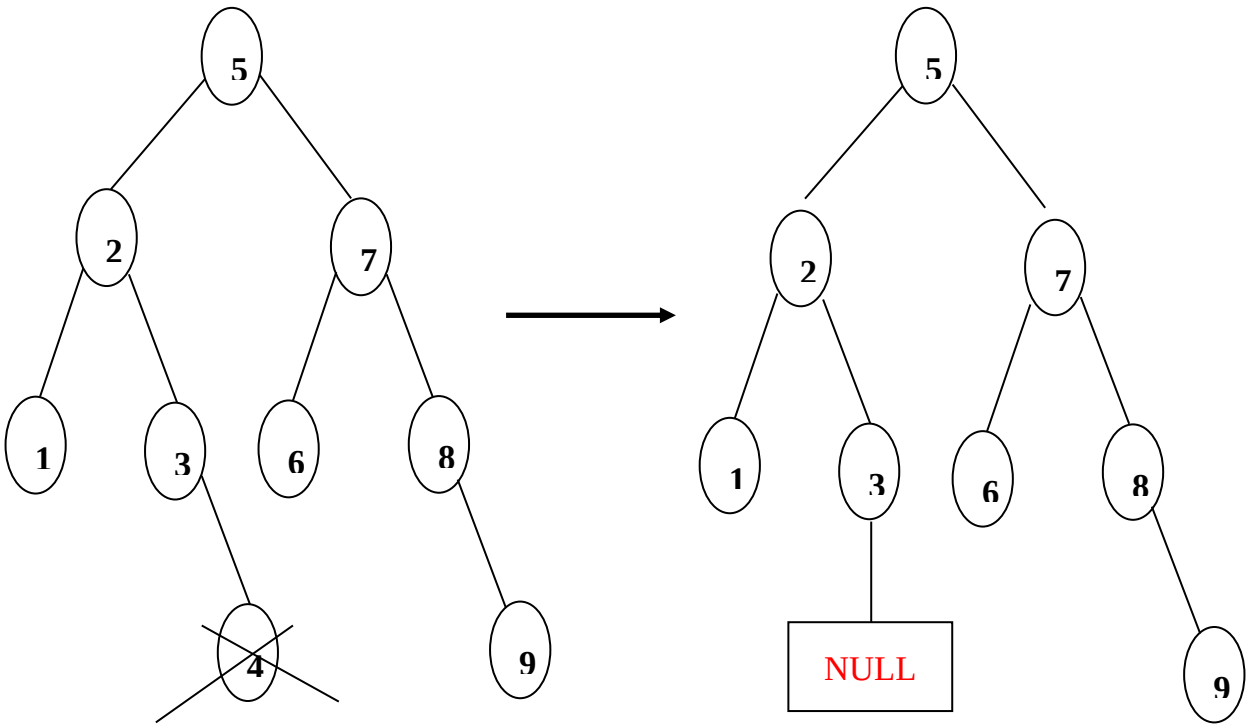
Trường hợp 3: Trước khi xóa X ta móc nối cha của X với con duy nhất của nó.

Ví dụ:

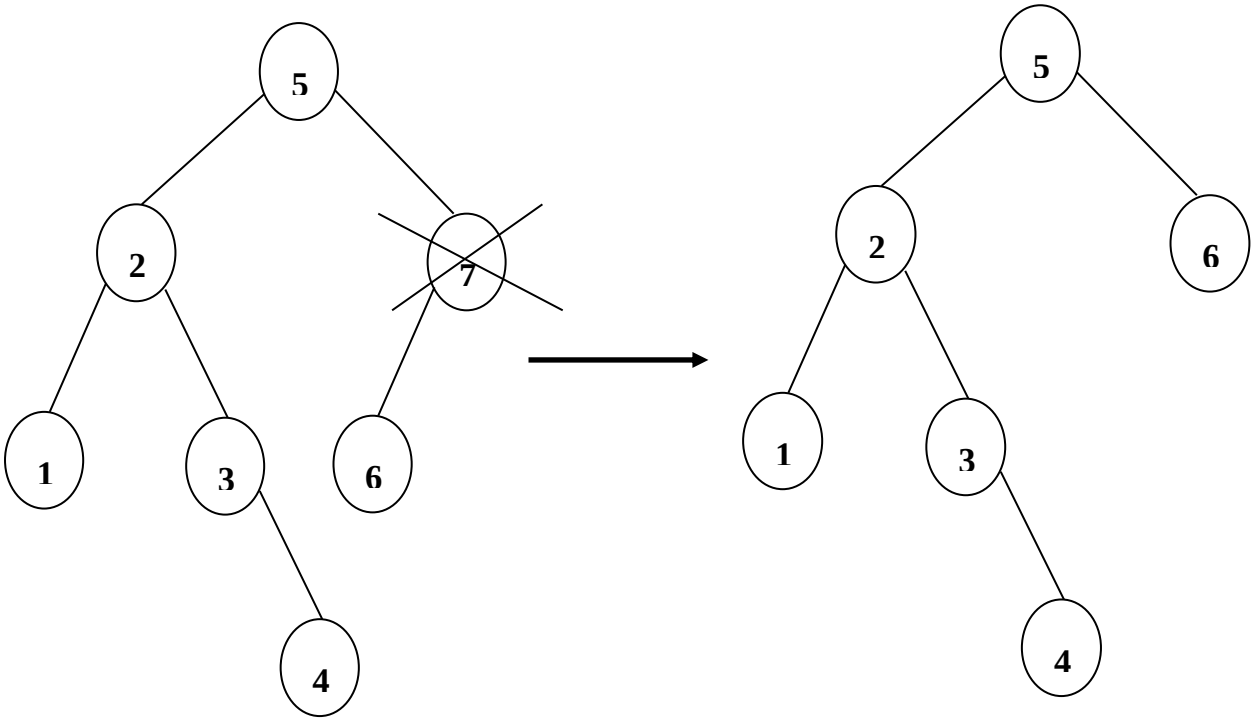
Trường hợp 1: **Xóa node có cả cây con trái và cây con phải** (Xóa node chứa khóa 2 trên cây nhị phân tìm kiếm sau)



Trường hợp 2: **Xóa node lá**



Trường hợp 3: **Xóa node chỉ chứa node con trái**



5.4. CÂY BIỂU THỨC

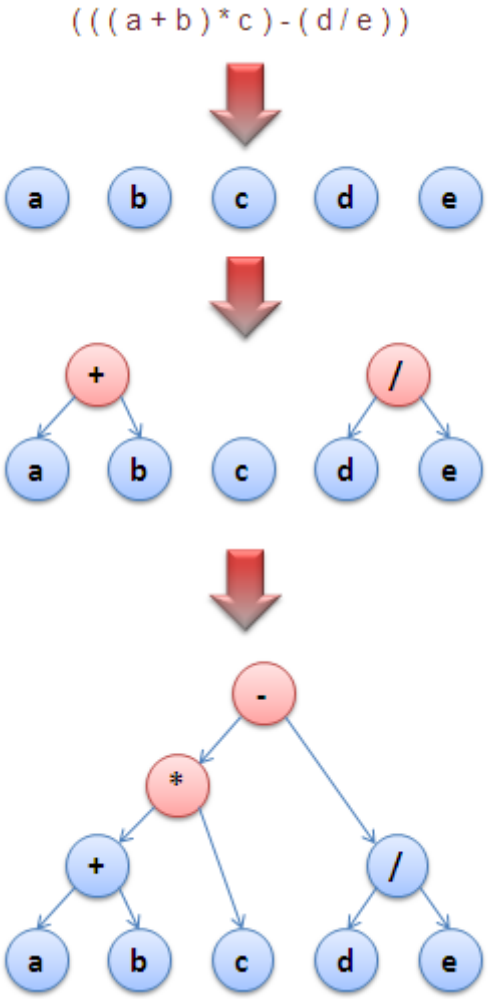
Cây biểu thức là cây nhị phân, có các lá là toán hạng hoặc hằng số, các nút không phải lá là các toán tử. (lưu ý các toán tử sin, cos chỉ cho phép số lượng nút con là 1 - toán tử 1 ngôi).

Ví dụ: Xét một biểu thức đại số đơn giản sau: **(a+b)*c-d/e.**

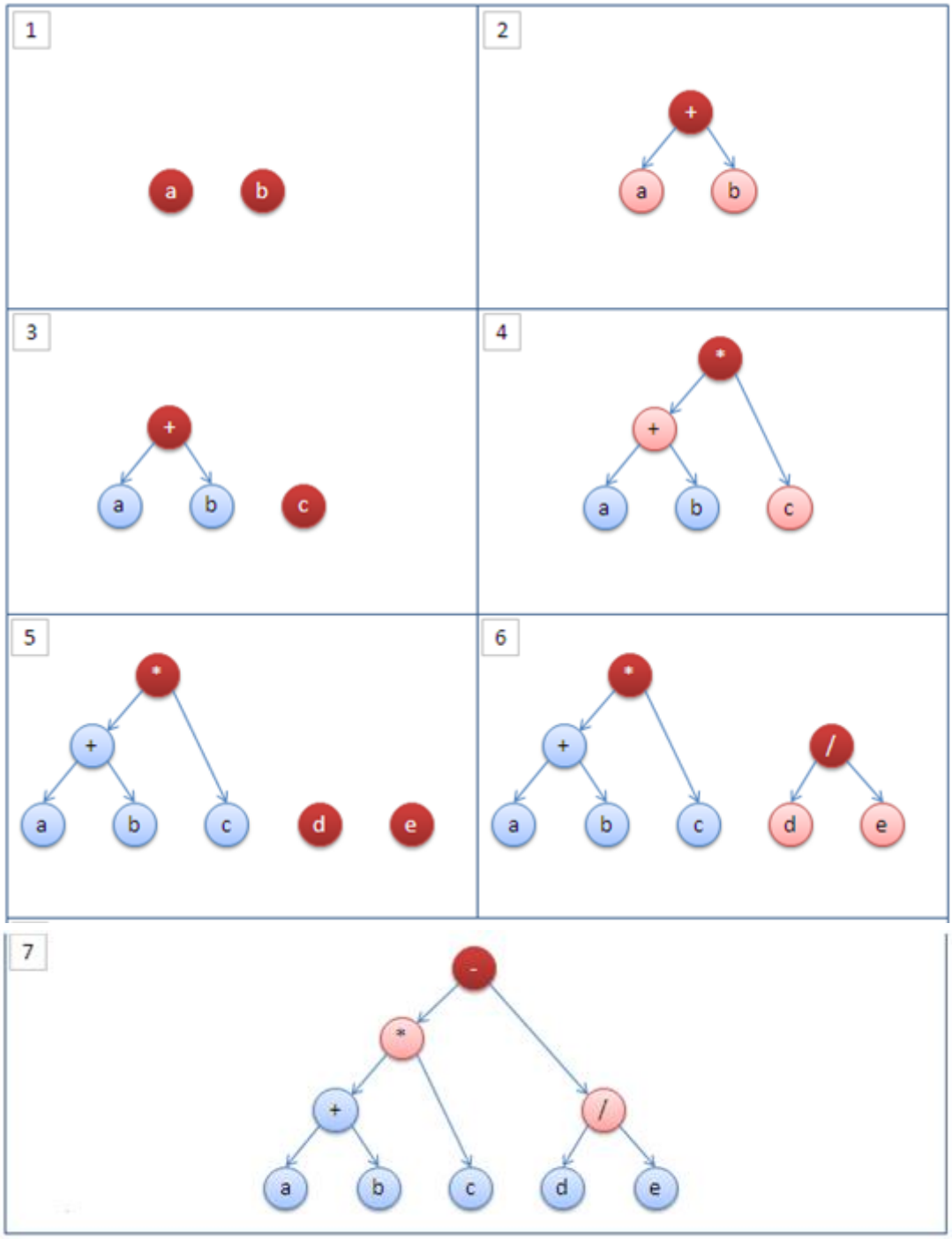
Dựa vào các quy tắc tính toán thông thường, ta có thể nhóm chúng lại bằng các cặp ngoặc đơn để thấy rõ hơn mức ưu tiên của chúng: **(((a+b)*c)-(d/e))**

Tất cả đều được phân cấp theo thứ tự ưu tiên một cách rõ ràng, dựa vào đó việc chuyển đổi biểu thức trên thành một cây biểu thức rất đơn giản như hình sau:

Ta thấy rằng cây biểu thức trên là một cây nhị phân vì các toán tử của nó là toán tử hai ngôi. Các node lá bao giờ cũng là toán hạng và các node còn lại phải là toán tử. Dựa vào độ sâu của các node này, ta biết được toán tử nào sẽ được thực hiện trước, tức là toán tử (+) nằm ở gốc của cây sẽ được thực hiện cuối cùng.



Minh họa quá trình xây dựng cây biểu thức với biểu thức đầu vào (a+b)*c-d/e như sau:



Quy tắc biểu diễn một biểu thức toán học trên cây như sau:

- Mỗi nút lá có nhãn biểu diễn cho một toán hạng.
- Mỗi nút trung gian biểu diễn một toán tử.

Giả sử nút n biểu diễn cho một toán tử hai ngôi θ (chẳng hạn + hoặc *), nút con bên trái biểu diễn cho biểu thức E1, nút con bên phải biểu diễn cho biểu thức E2 thì nút n biểu diễn biểu thức E1θ E2. Nếu θ là phép toán một ngôi thì nút chứa phép toán θ chỉ có một nút con, nút con này biểu diễn cho toán hạng của θ.

Khi chúng ta duyệt một cây biểu diễn một biểu thức toán học và liệt kê nhãn của các nút theo thứ tự duyệt thì ta có:

- Biểu thức dạng tiền tố (prefix) tương ứng với phép duyệt tiền tự của cây.
- Biểu thức dạng trung tố (infix) tương ứng với phép duyệt trung tự của cây.
- Biểu thức dạng hậu tố (posfix) tương ứng với phép duyệt hậu tự của cây.

Sau đây là chương trình tính giá trị biểu thức được lưu trữ dưới dạng cây nhị phân:

```
#include <iostream.h>
#include <conio.h>
struct NUT
{
  char x;
  NUT *Left,*Right;
};
typedef NUT* Nut;
char *st="*+23-74";
int i=-1;
char Token;
Nut Root=NULL;
void TaoCay(char *st, Nut &Root)
{
  i++;
  Token = st[i];
  Root = new(NUT);
  Root->x = Token;
  if((Token<='9')&&(Token>='0'))
  {
    Root->Left = NULL;
    Root->Right = NULL;
  }
  else  //Token is [+,-,*,/]
    {
      TaoCay(st,Root->Left);
      TaoCay(st,Root->Right);
    }
}
```

```
void XemCay(Nut Root,int x,int y, int k)
{
  if(Root)
  {
    gotoxy(x,y); cprintf("%c",Root->x);
    XemCay(Root->Left,x - k/2,y+2,k-3);
    XemCay(Root->Right,x + k/2,y+2,k-3);
  }
}
float Value(Nut Root)
{
  switch(Root->x)
  {
    case '+':return Value(Root->Left) + Value(Root->Right);
    case '-':return Value(Root->Left) - Value(Root->Right);
    case '*':return Value(Root->Left) * Value(Root->Right);
    case '/':return Value(Root->Left) / Value(Root->Right);
    default : return Root->x - 48;
  }
}
main()
{
  TaoCay(st,Root);
  XemCay(Root,40,5,20);
  getch();
  cout<<"Gia tri bieu thuc: "<<Value(Root);
}
```

BÀI TẬP

Bài 1: Cho biểu thức số học dạng trung tố.

1. Tổ chức cây nhị phân để lưu trữ biểu thức số học.
2. Viết hàm để tạo cây nhị phân từ biểu thức số học.
3. Viết hàm để tính giá trị của biểu thức số học từ cây nhị phân.

Bài 2: Cho cây tìm kiếm nhị phân được khai báo như sau

```
struct NUT
{
 int  Gtri;
 NUT  *Trai, *Phai;
};
typedef NUT* BST;
```

1. Viết hàm **INSERT(int X, BST &Goc)** để chèn một nút có giá trị X vào cây.

2. Viết hàm **NHAP(BST &Goc)** để nhập vào một số nút cho cây nhị phân.

3. Viết hàm **DUYET(BST Goc)** để duyệt cây và in các giá trị của các nút ra màn hình theo thứ tự giữa.

4. Viết hàm **int TONG(BST Goc)** để tính tổng giá trị các nút của cây nhị phân.

5. Viết hàm **int DEM(BST Goc)** để đếm xem cây có bao nhiêu nút lá.

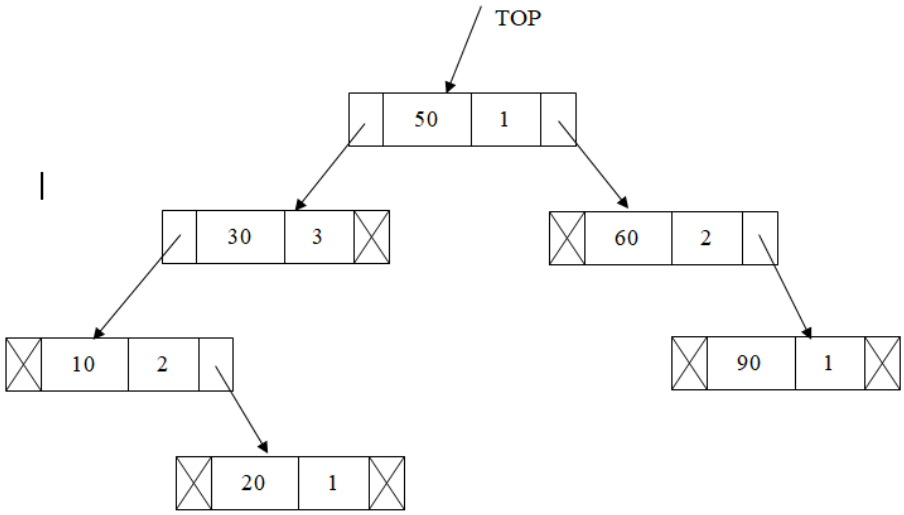
6. Viết hàm **int CayTK(BST Goc)** để kiểm tra xem cây có nút gốc là goc có phải là cây nhị phân tìm kiếm hay không?

7. Viết hàm **BST COPY(BST Goc)** để tạo ra một bảng sao của cây nhị phân có nút gốc được trỏ bởi Goc.

8. Viết hàm **DELETE(int X, BST &Goc)** để xoá nút trên cây nhị phân có giá trị X.

Bài 3: Người ta dùng một cây nhị phân tìm kiếm để lưu trữ dãy các số nguyên mà mỗi nút của cây gồm 4 trường sau: 2 trường liên kết TRAI và PHAI; 2 trường còn lại là GIATRI và SOLUONG. Trường GIATRI là trường khóa lưu giá trị nguyên trong dãy, trường SOLUONG lưu số lần xuất hiện của giá trị khóa tương ứng trong dãy.

Ví dụ: Với dãy 50, 30, 60, 10, 30, 20, 10, 90, 60, 30 sẽ được lưu trữ như sau:



1. Hãy xây dựng cấu trúc dữ liệu cho cây nhị phân nói trên.
2. Viết hàm để duyệt cây theo thứ tự giữa với trường khóa là GIATRI.
3. Viết hàm để bổ sung một giá trị X vào cây nhị phân.
4. Viết hàm để duyệt cây theo trường khoá GIATRI sao cho các giá trị in ra theo thứ tự giảm dần.

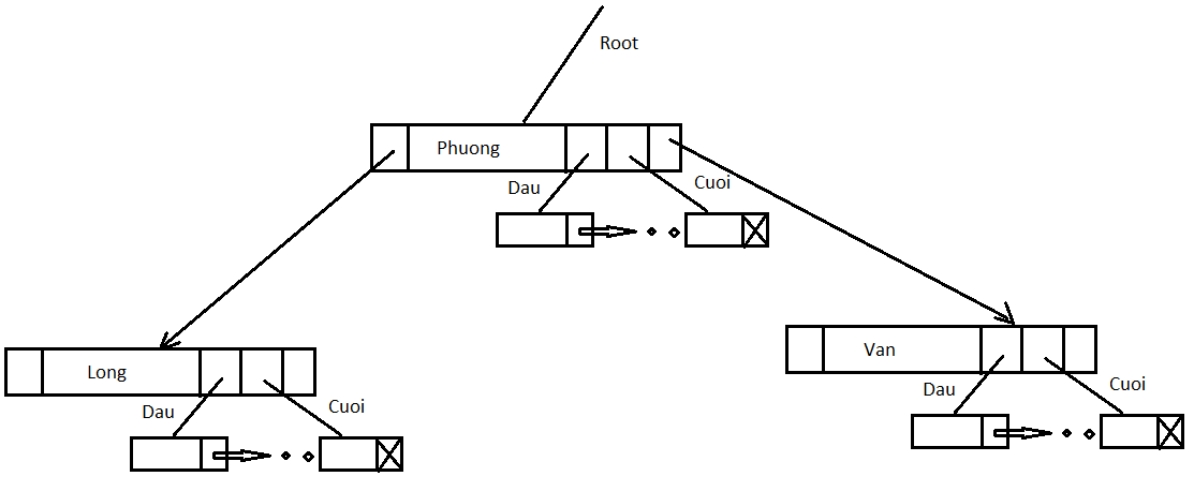
Bài 4: QUẢN LÝ THƯ VIỆN

Người ta quản lý các thông tin của một thư viện dưới dạng cây BST với khóa là *TenTG* (tên tác giả). Mỗi nút của cây là một cấu trúc gồm trường *TenTG* và 4 trường con trỏ: 2 con trỏ *Trái* và *Phai* lần lượt trỏ tới các nút con trái và nút con phải, 2 con trỏ *Dau* và *Cuoi* lần lượt trỏ tới phần tử đầu tiên và cuối cùng của một danh sách liên kết đơn dùng để ghi nhận các sách có trong thư viện của tác giả đó. Mỗi phần tử của danh sách liên kết là một cấu trúc gồm 2 trường: *TenSach* (tên sách) và *Tieptheo* (chứa địa chỉ nút tiếp theo).

Người ta khai báo CTDL cho bài toán trên như sau:

```
struct SACH
{
     string TenSach;
     SACH *Tieptheo;
};
typedef SACH* TroSach;
struct TACGIA
{
     string TenTG;
     TroSach Dau, Cuoi;
     TACGIA *Trai, *Phai;
```

```
};
typedef TACGIA* TroTG;
```



1. Viết hàm **TroTG Search(string Name, TroTG Root)** để tìm đến nút trên cây BST có nút gốc được trỏ bởi con trỏ Root chứa tác giả có tên là *Name*. Nếu không tìm thấy, hàm trả về giá trị NULL.
2. Viết hàm **void Insert(string Title, string Name, TroTG &Root)** để bổ sung tác giả có tên *Name* với cuốn sách có tiêu đề *Title* vào cây BST có nút gốc được trỏ bởi con trỏ *Root* theo cách sau:
   - Nếu *Name* và *Title* đã có trong thư viện thì không làm gì nữa.
   - Nếu *Name* đã có và *Title* chưa có thì bổ sung *Title* vào cuối danh sách liên kết đơn tương ứng với nút có *TenTG = Name*.
   - Nếu *Name* chưa có thì bổ sung một nút mới vào cây BST với *TenGT = Name* và *TenSach = Title*.
3. Viết hàm **void Duyet(TroTG Root)** để hiển thị lên màn hình tất cả các tác giả và các tiêu đề sách của các tác giả đó.
4. Viết hàm **int Dem(TroTG Root)** để đếm số tác giả viết từ 2 cuốn sách trở lên.

5. Hãy xây dựng cấu trúc dữ liệu cho cây nhị phân nói trên.
6. Viết hàm để duyệt cây theo thứ tự giữa với trường khóa là GIATRI.
7. Viết hàm để bổ sung một giá trị X vào cây nhị phân.
8. Viết hàm để duyệt cây theo trường khoá GIATRI sao cho các giá trị in ra theo thứ tự giảm dần.

TÀI LIỆU THAM KHẢO

[1]. Đỗ Xuân Lôi, *Cấu trúc dữ liệu và giải thuật*, NXB Thống kê, 1994.

[2]. Nguyễn Trung Trực, *Cấu trúc dữ liệu*, NXB KHKT, 1996.

[3]. Jon Bentley, *Những viên ngọc trong lập trình*, NXB Thống kê, 2002

[4]. Dinesh P. Mehta and Sartaj Sahni, *Data structures and Applications*, Chapman & Hall/CRC, 2005.

[5]. Kenneth H. Rosen, *Toán học rời rạc ứng dụng trong tin học*, NXB KHKT, 2003.