

MỤC LỤC

Chương 1: GIỚI THIỆU CHUNG.....	3
1.1. Thuật toán và cấu trúc dữ liệu:.....	3
1.2. Một số vấn đề liên quan:.....	3
1.3. Ngôn ngữ diễn đạt thuật toán:.....	3
1.3.1. Cấu trúc của một chương trình chính:.....	3
1.3.2. Các ký tự:.....	4
1.3.3. Các câu lệnh:.....	4
1.3.4. Chương trình con:.....	5
CHƯƠNG 2: THIẾT KẾ VÀ PHÂN TÍCH GIẢI THUẬT.....	7
2.1. Thiết kế thuật toán:.....	7
2.1.1. Module hoá thuật toán:.....	7
2.1.2. Phương pháp tinh chỉnh từng bước:.....	8
2.2. Phân tích thuật toán:.....	8
2.2.1. Tính đúng đắn:.....	8
2.2.2. Mâu thuẫn giữa tính đơn giản và tính hiệu quả:.....	8
2.2.3. Phân tích thời gian thực hiện thuật toán:.....	8
CHƯƠNG 3: ĐỆ QUY (RECURSION).....	11
3.1. Đại cương:.....	11
3.2. Phương pháp để thiết kế một thuật toán đệ quy:.....	12
3.3. Thuật toán quay lui:.....	15
CHƯƠNG 4: MẢNG VÀ DANH SÁCH TUYẾN TÍNH.....	17
4.1. Mảng và cấu trúc lưu trữ của mảng:.....	17
4.2. Danh sách tuyến tính (Linear list):.....	18
4.3. Ngăn xếp (Stack):.....	19
4.3.1. Định nghĩa:.....	19
4.3.2. Lưu trữ Stack bằng mảng:.....	19
4.3.3. Các ví dụ:.....	20
4.3.4. Stack với việc cài đặt thuật toán đệ quy:.....	23
4.4. Hàng đợi (Queue):.....	25
4.4.1. Định nghĩa:.....	25
4.4.2. Lưu trữ Queue bằng mảng:.....	25
CHƯƠNG 5: DANH SÁCH MÓC NỐI (LINKED LIST).....	28
5.1. Danh sách móc nối đơn:.....	28
5.1.1. Tổ chức danh sách nối đơn:.....	28
5.1.2. Một số phép toán trên danh sách nối đơn:.....	28
5.2. Danh sách nối vòng:.....	30
5.2.1. Nguyên tắc:.....	30
5.2.2. Thuật toán bổ sung và loại bỏ một nút của danh sách nối vòng:.....	31
5.3. Danh sách nối kép:.....	31
5.3.1. Tổ chức:.....	31

5.3.2. Một số phép toán trên danh sách nối kép:.....	31
5.4. Ví dụ về việc sử dụng danh sách móc nối:.....	32
5.5. Stack và Queue móc nối:.....	34
CHƯƠNG 6: CÂY (TREE).....	37
6.1. Định nghĩa và các khái niệm:.....	37
6.1.1. Định nghĩa:.....	37
6.1.2. Các khái niệm liên quan:.....	37
6.2. Cây nhị phân:.....	38
6.2.1. Định nghĩa và tính chất:.....	38
6.2.2. Biểu diễn cây nhị phân:.....	39
6.2.3. Phép duyệt cây nhị phân:.....	40
6.2.4. Cây nhị phân nối vòng:.....	45
6.3. Cây tổng quát:.....	47
6.3.1. Biểu diễn cây tổng quát:.....	47
6.3.2. Phép duyệt cây tổng quát:.....	48
6.4. Ứng dụng (Biểu diễn cây biểu thức số học):.....	49
CHƯƠNG 7: ĐỒ THỊ (GRAPH).....	53
7.1. Định nghĩa và các khái niệm về đồ thị:.....	53
7.2. Biểu diễn đồ thị:.....	54
7.2.1. Biểu diễn bằng ma trận lân cận (ma trận kề):.....	54
7.2.2. Biểu diễn bằng danh sách lân cận (danh sách kề):.....	54
7.3. Phép duyệt một đồ thị:.....	55
7.3.1. Tìm kiếm theo chiều sâu:.....	55
7.3.2. Tìm kiếm theo chiều rộng:.....	56
7.4. Cây khung và cây khung với giá cực tiểu:.....	57
CHƯƠNG 8: SẮP XẾP.....	59
8.1. Đặt vấn đề:.....	59
8.2. Một số phương pháp sắp xếp đơn giản:.....	59
8.2.1. Sắp xếp kiểu lựa chọn:.....	59
8.2.2. Sắp xếp kiểu chèn:.....	59
8.2.3. Sắp xếp kiểu nổi bọt:.....	60
8.3. Sắp xếp kiểu phân đoạn (Sắp xếp nhanh - quick sort):.....	60
8.4. Sắp xếp kiểu vun đống (Heap sort):.....	61
8.5. Sắp xếp kiểu trộn (Merge sort):.....	62
CHƯƠNG 9: TÌM KIẾM.....	64
9.1. Bài toán tìm kiếm:.....	64
9.2. Tìm kiếm tuần tự:.....	64
9.3. Tìm kiếm nhị phân:.....	64
9.4. Cây nhị phân tìm kiếm:.....	64
TÀI LIỆU THAM KHẢO.....	67

CHƯƠNG 1: GIỚI THIỆU CHUNG

1.1. Thuật toán và cấu trúc dữ liệu:

Theo Niklaus Wirth: Thuật toán + Cấu trúc dữ liệu = Chương trình.

Ví dụ: Cho 1 dãy các phần tử, có thể biểu diễn dưới dạng mảng hoặc danh sách.

Cấu trúc dữ liệu và thuật toán có mối quan hệ mật thiết với nhau. do đó việc nghiên cứu các cấu trúc dữ liệu sau này đi đôi với việc xác lập các thuật toán xử lý trên các cấu trúc ấy.

1.2. Một số vấn đề liên quan:

Lựa chọn một cấu trúc dữ liệu thích hợp để tổ chức dữ liệu vào ra và trên cơ sở đó xây dựng được thuật toán xử lý hữu hiệu nhằm đưa tới kết quả mong muốn cho bài toán là một khâu rất quan trọng.

Ta cần phân biệt 2 loại quy cách dữ liệu:

- Quy cách biểu diễn hình thức: Còn được gọi là cấu trúc logic của dữ liệu. Đối với mỗi ngôn ngữ lập trình xác định sẽ có một bộ cấu trúc logic của dữ liệu. Dữ liệu thuộc loại cấu trúc nào thì cần phải có mô tả kiểu dữ liệu tương ứng với cấu trúc dữ liệu đó. Ví dụ: Trong C có các kiểu dữ liệu: Struct, Union, File,...
- Quy cách lưu trữ: là cách biểu diễn một cấu trúc dữ liệu trong bộ nhớ. Ví dụ: Cấu trúc dữ liệu mảng được lưu trữ trong bộ nhớ theo quy tắc lưu trữ kế tiếp. Có 2 quy cách lưu trữ:

Lưu trữ trong: ví dụ RAM.

Lưu trữ ngoài: ví dụ đĩa (disk).

1.3. Ngôn ngữ diễn đạt thuật toán:

Ngôn ngữ diễn đạt thuật toán được quy ước sử dụng trong giáo trình này là ngôn ngữ tựa C++.

Đặc điểm: Gần giống với Turbo C++, do đó dễ dàng trong việc chuyển một chương trình viết bằng ngôn ngữ tựa C++ sang ngôn ngữ C++.

1.3.1. Cấu trúc của một chương trình chính:

```
void main()
```

```
{
```

```
    S1;
```

```
    S2;
```

} Các lệnh của chương trình dùng để diễn tả thuật

```
    ...  
    Sn;  
}
```

Lưu ý:

- Để đơn giản, chương trình có thể không cần viết khai báo. Tuy nhiên có thể mô tả trước chương trình bằng ngôn ngữ tự nhiên.
- Phần thuyết minh được đặt giữa 2 dấu /* , */ hoặc // để ghi chú trên 1 dòng.

Ví dụ:

```
void main()    /* Chương trình chuyển số hệ 10 thành hệ 2 */
{
    cout << "n = ";
    cin >> n;    /* Nhập n là số hệ cơ số 10 */
    T=0;
    while (n!=0)
    {
        r = n % 2;
        Push(T, r);
        n = n / 2;
    }
    cout << "Kết quả chuyển đổi sang hệ cơ số 2 là: ";
    while (T!=0)
    {
        Pop(T, r);
        cout << r;
    }
}
```

1.3.2. Các ký tự:

Các ký tự sử dụng trong chương trình là tương tự như trong C++.
Lưu ý: Trong C++ là có sự phân biệt giữa chữ hoa và chữ thường.

1.3.3. Các câu lệnh:

- **Lệnh gán:** $V = E$;

Trong đó: V là biến (variable), và E là biểu thức (expression).

Lưu ý: Có thể dùng phép gán chung. Ví dụ: $a=b=1$;

- **Lệnh ghép:** $\{S_1; S_2; \dots; S_n\}$ coi như là một câu lệnh (trong đó S_i là các câu lệnh).

- **Lệnh if:** Tương tự như lệnh if của ngôn ngữ C.

$\text{if} (<\text{biểu thức điều kiện}>) <\text{câu lệnh}>;$

hoặc: $\text{if} (<\text{biểu thức điều kiện}>) <\text{câu lệnh 1}>;$

$\text{else} <\text{câu lệnh 2}>;$

- **Lệnh switch:** Theo cấu trúc sau:

$\text{switch} (<\text{biểu thức}>)$

```

{
    case gt1: S1;
    case gt2: S2;
    ...
    case gtn: Sn;
    [default : Sn+1;]
}

```

- *Lệnh lặp*: for, while, do ... while: Tương tự như các lệnh lặp của C.
- *Lệnh nhảy*: goto n (n: số hiệu/nhãn của chương trình).
- *Lệnh vào ra*: cin và cout giống như C++.

1.3.4. Chương trình con:

```

<kiểu trả về> <Tên hàm>(<danh sách tham số>)
{
    S1;
    S2;
    ...
    S3;
    [return (giá trị trả về) ] → Báo kết thúc chương trình con
}

```

Lưu ý: Nếu hàm có kiểu trả về khác kiểu void thì khi kết thúc hàm phải có câu lệnh **return <giá trị của hàm>** để gán kết quả cho hàm.

Sau đây là ví dụ về hàm có trả về giá trị.

Ví dụ: Viết chương trình con dạng hàm NamNhuan(x). Cho kết quả nếu số x là năm nhuận có giá trị là True(1), ngược lại có giá trị là False(0); chẳng hạn: NamNhuan(1996) cho giá trị 1, NamNhuan(1997) cho giá trị 0. Biết rằng x được gọi là năm nhuận nếu x chia hết cho 4 và x không chia hết cho 100 hoặc x chia hết cho 400.

Cách 1: int namnhuan(x)

```

{ if ((x % 4 == 0 && x % 100 != 0) || (x % 400 == 0))
    return 1;
    else
        return 0;
}

```

Cách 2: int namnhuan(x)

```

{ return(((x % 4 == 0) && (x % 100 != 0)) ||

```

```
        (x % 400 == 0));  
    }
```

Ví dụ viết về chương trình con không có giá trị trả về (hay còn gọi là thủ tục).

Ví dụ: Viết hàm Hoandoi(a, b) để hoán đổi giá trị của 2 biến số a và b cho nhau.

Cách 1:

```
void hoandoi(&a, &b) //a và b là các tham biến  
{ tam=a;  
  a=b;  
  b=tam;  
}
```

Cách 2:

```
void hoandoi(&a, &b)  
{ a= a+b;  
  b= a-b;  
  a= a-b;  
}
```

Lưu ý: Bên trong 1 chương trình con có thể dùng lệnh return; (thoát khỏi chương trình con), exit(1) (thoát khỏi chương trình chính).

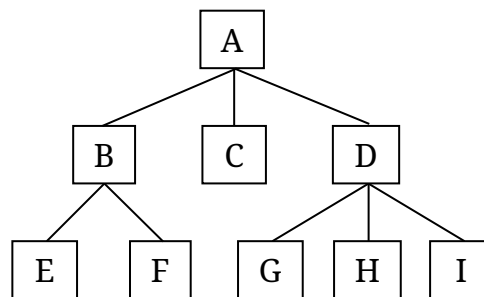
CHƯƠNG 2: THIẾT KẾ VÀ PHÂN TÍCH THUẬT TOÁN

2.1. Thiết kế thuật toán:

2.1.1. Module hoá thuật toán:

Các bài toán ngày càng đa dạng và phức tạp, do đó thuật toán mà ta đề xuất càng có quy mô lớn và việc viết chương trình cần có một lượng lập trình đồng đảo. Muốn làm được việc này, người ta phân chia các bài toán lớn thành các bài toán nhỏ (module). Và dĩ nhiên một module có thể chia nhỏ thành các module con khác nữa,... bấy giờ việc tổ chức lời giải sẽ được thể hiện theo một cấu trúc phân cấp.

Ví dụ:



Quá trình module hoá bài toán được xem là nguyên lý “chia để trị” (divide & conquer) hay còn gọi là thiết kế từ đỉnh xuống (top-down) hoặc là thiết kế từ khái quát đến chi tiết (specialization).

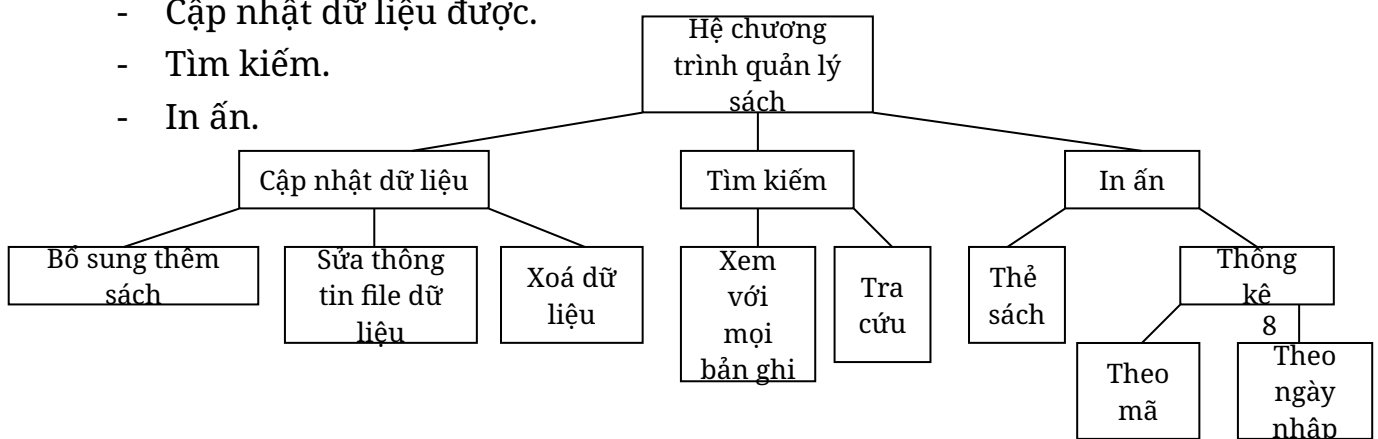
Việc module hoá trong lập trình thể hiện ở:

- Các chương trình con.
- Cụm các chương trình con xung quanh một cấu trúc dữ liệu nào đó. Chẳng hạn, thư viện trong C.

Ví dụ: Chương trình quản lý đầu sách của một thư viện nhằm phục vụ độc giả tra cứu sách. Cụ thể, giả sử ta đã có một file dữ liệu gồm các bảng ghi về các thông tin liên quan đến một đầu sách như: tên sách, mã số, tác giả, nhà xuất bản, năm xuất bản, giá tiền, ...

Yêu cầu:

- Cập nhật dữ liệu được.
- Tìm kiếm.
- In ấn.



Nhận xét:

- Việc module hoá làm cho bài toán được định hướng rõ ràng.
- Bằng cách này, người ta có thể phân chia công việc cho đội ngũ lập trình.
- Đây là một công việc mất nhiều thời gian.

2.1.2. Phương pháp tinh chỉnh từng bước:

Phương pháp tinh chỉnh từng bước là phương pháp thiết kế thuật toán gắn liền với lập trình. Nó phản ánh tinh thần của quá trình module hoá và thiết kế thuật toán theo kiểu top-down.

Xuất phát từ ngôn ngữ tự nhiên của thuật toán, thuật toán sẽ được chi tiết hoá dần dần và cuối cùng công việc xử lý sẽ được thay thế dần bởi các câu lệnh (của một ngôn ngữ lập trình nào đó). Quá trình này là để trả lời dần dần các câu hỏi: What? (làm gì?), How (làm như thế nào?)

2.2. Phân tích thuật toán:

Chất lượng của một chương trình hay thuật toán bao gồm:

- Tính đúng đắn.
- Tính đơn giản (dễ hiểu, dễ quản lý, dễ lập).
- Tính tối ưu (hiệu quả) về mặt thời gian cũng như không gian nhớ.

2.2.1. Tính đúng đắn:

Đây là một yêu cầu phân tích quan trọng nhất cho một thuật toán. Thông thường, người ta thử nghiệm (test) nhờ một số bộ dữ liệu nào đó để cho chạy chương trình rồi so sánh kết quả thử nghiệm với kết quả mà ta đã biết. Tuy nhiên, theo Dijkstra: “Việc thử nghiệm chương trình chỉ chứng minh sự có mặt của lỗi chứ không chứng minh sự vắng mặt của lỗi”.

Ngày nay, với các công cụ toán học người ta có thể chứng minh tính đúng đắn của một thuật toán.

2.2.2. Mâu thuẫn giữa tính đơn giản và tính hiệu quả:

Một thuật toán đơn giản (dễ hiểu) chưa hẳn tối ưu về thời gian và bộ nhớ. Đối với những chương trình chỉ dùng một vài lần thì tính đơn giản có thể coi trọng nhưng nếu chương trình được sử dụng nhiều lần (ví dụ, các phần mềm) thì thời gian thực hiện rõ ràng phải được chú ý.

Yêu cầu về thời gian và không gian ít khi có một giải pháp trọn vẹn.

2.2.3. Phân tích thời gian thực hiện thuật toán:

Thời gian thực hiện thuật toán phụ thuộc vào nhiều yếu tố:

- Kích thước dữ liệu đưa vào (dung lượng). Nếu gọi n là kích thước dữ liệu vào thì thời gian thực hiện một thuật toán, ký hiệu là $T(n)$.
- Tốc độ xử lý của máy tính, bộ nhớ (RAM).
- Ngôn ngữ để viết chương trình.

Tuy nhiên, ta có thể so sánh thời gian thực hiện của hai thuật toán khác nhau.

Ví dụ: Nếu thời gian thực hiện của thuật toán thứ nhất $T_1(n) = Cn^2$ (C : hằng dương) và thời gian thực hiện thuật toán thứ hai $T_2(n) = Kn$ (K : hằng) thì khi n khá lớn, thời gian thực hiện thuật toán 2 sẽ tối ưu hơn so với thuật toán 1.

Cách đánh giá thời gian thực hiện thuật toán theo kiểu trên được gọi là đánh giá thời gian thực hiện thuật toán theo “độ phức tạp tính toán của thuật toán”.

2.2.3.1. Độ phức tạp tính toán của thuật toán:

Nếu thời gian thực hiện một thuật toán là $T(n) = Cn^2$ (C : hằng), thì ta nói rằng: Độ phức tạp tính toán của thuật toán này có cấp là n^2 và ta ký hiệu $T(n) = O(n^2)$.

Tổng quát: $T(n) = O(g(n))$ thì ta nói độ phức tạp của thuật toán có cấp là $g(n)$.

2.2.3.2. Xác định độ phức tạp của thuật toán:

Việc xác định độ phức tạp tính toán của một thuật toán nói chung là phức tạp. Tuy nhiên, trong thực tế độ phức tạp của một thuật toán có thể được xác định từ độ phức tạp từng phần của thuật toán. Cụ thể, ta có một số quy tắc sau:

- Quy tắc tính tổng:

Nếu chương trình P được phân tích thành 2 phần: P_1, P_2 và nếu độ phức tạp của P_1 là $T_1(n) = O(g_1(n))$ và độ phức tạp của P_2 là $T_2(n) = O(g_2(n))$ thì độ phức tạp của P là: $T(n) = O(\max(g_1(n), g_2(n)))$.

Ví dụ: $g_1(n) = n^2, g_2(n) = n^3$. Suy ra: $T(n) = O(n^3)$.

Lưu ý: $g_1(n) \leq g_2(n) \quad (\forall n \geq n_0) \Rightarrow O(g_1(n) + g_2(n)) = O(g_2(n))$

Ví dụ: $O(n + \log_2 n) = O(n)$

- *Quy tắc nhân:*

Nếu độ phức tạp của P_1 là $O(g_1(n))$, độ phức tạp của P_2 là $O(g_2(n))$ thì độ phức tạp của P_1 lồng P_2 (P_1 câu lệnh lặp) thì độ phức tạp tính toán là $O(g_1(n).g_2(n))$.

Lưu ý:

- Câu lệnh gán, cin, cout, if, switch có thời gian thực hiện bằng hằng số $C = O(1)$.
- Câu lệnh lặp trong vòng $g(n)$ lần thì sẽ có thời gian thực hiện là $O(g(n))$.
- $O(Cg(n)) = O(g(n))$ (C : hằng)

Ví dụ:

```
1) Câu lệnh:  for (i=1;i<=n;i++)          // O(n)
                P=P*i;                      // O(1)
```

có thời gian thực hiện là: $O(n*1) = O(n)$.

```
2) for (i=1;i<=n;i++)
    for (j=1;j<=n;j++) x=x+1;
```

có thời gian thực hiện là: $O(n*n*1) = O(n^2)$.

- Thông thường, để xác định độ phức tạp tính toán của một thuật toán, người ta đi tìm một lệnh/phép toán có số lần thực hiện là nhiều nhất (lệnh/phép toán tích cực) từ đó tính số lần này \Rightarrow độ phức tạp của tính toán.
- Có khi thời gian thực hiện một thuật toán còn phụ thuộc vào đặc điểm của dữ liệu. Bấy giờ $T(n)$ trong trường hợp thuận lợi nhất có thể khác $T(n)$ trong trường hợp xấu nhất. Tuy nhiên, thông thường người ta vẫn đánh giá độ phức tạp tính toán của thuật toán thông qua $T(n)$ trong trường hợp xấu nhất.

Ví dụ: Cho một dãy gồm có n phần tử mảng: $V[0], V[1], \dots, V[n-1]$. X là một giá trị cho trước.

```
void timkiem(x)
{
    found=0; // gán giá trị cho found lúc ban đầu là False
    i=0;
    while ((i< n) && (!found))
        if (v[i]==x)
```

```

        {
            cout << i; found=1;
        }
        else i=i+1;
    if (found==0)
        cout << "không có";
}

```

$\Rightarrow T(n) \text{ thuận lợi} = O(1)$ $(X = V[0])$
 $T(n) \text{ xấu nhất} = O(n)$ $(X \neq V[i], \forall i=0..n-1)$
 $\Rightarrow T(n) = O(n)$

CHƯƠNG 3: ĐỆ QUY (RECURSION)

3.1. Đại cương:

- Chương trình đệ quy là chương trình gọi đến chính nó.

Ví dụ: Một hàm đệ quy là một hàm được định nghĩa dựa vào chính nó.

- Trong lý thuyết tin học, người ta thường dùng thủ thuật đệ quy để định nghĩa các đối tượng.

Ví dụ: Tên biến được định nghĩa như sau:

- Mỗi chữ cái là một tên.
- Nếu t là tên biến thì t <chữ cái>, t <chữ số> cũng là tên biến.

- Một chương trình đệ quy hoặc một định nghĩa đệ quy thì không thể gọi đến chính nó mãi mãi mà phải có một điểm dừng đến một trường hợp đặc biệt nào đó, mà ta gọi là trường hợp suy biến (degenerate case).

Ví dụ: Cho số tự nhiên n, ta định nghĩa n! như sau:
$$n! = \begin{cases} n * (n-1)! \\ 0! = 1 \end{cases}$$

- Lời giải đệ quy: Nếu lời giải của một bài toán T nào đó được thực hiện bằng một lời giải của bài toán T' có dạng giống như T, nhưng theo một nghĩa nào đó T' là "nhỏ hơn" T và T' có khuynh hướng ngày càng tiếp cận với trường hợp suy biến.

Ví dụ: Cho dãy các phần tử mảng V[1], V[2], ..., V[n] đã được sắp xếp theo thứ tự tăng dần, gọi X là một giá trị bất kỳ. Viết thuật toán tìm kiếm để in vị trí của phần tử nào đó trong mảng có giá trị bằng X (nếu có). Ngược lại, thông báo không có.

```
void timkiem(d, c, x)
{
    if (d>c)
        cout << "khong co";
    else
    {
        g=(d+c)/ 2;
        if (x==V[g])
            cout << g;
        else if (x<V[g]) timkiem(d, g-1, x);
    }
}
```

```

        else timkiem(g+1, c, x);
    }
}

```

Nhận xét:

- Bài toán tìm kiếm ban đầu được tách thành các bài toán tìm kiếm với phạm vi nhỏ hơn cho đến khi gặp phải các trường hợp suy biến. Chính việc phân tích đó, người ta đã xem thuật toán đệ quy là thuật toán thể hiện phương pháp "chia để trị".
- Nếu thủ tục hoặc hàm chứa lời gọi đến chính nó (ví dụ trên) thì được gọi là đệ quy trực tiếp. Ngược lại, có thủ tục chứa lời gọi đến thủ tục khác mà ở thủ tục này chứa lại lời gọi đến nó thì được gọi là đệ quy gián tiếp, hay còn gọi là đệ quy tương hỗ hay còn gọi là Forward.

Ví dụ:

```

void Ba(int n)
{
    cout << n;
    if (n>0) Ong(n-1);
}

void Ong(int n);
{
    cout << n;
    if (n>0) Ba(n-1);
}

void main()
{
    Ong(3);
}

```

Kết quả:

3	Ong
2	Ba
1	Ong
0	Ba

3.2. Phương pháp để thiết kế một thuật toán đệ quy:

- Tham số hoá bài toán.

- Phân tích trường hợp chung (đưa bài toán dưới dạng bài toán cùng loại nhưng có phạm vi giải quyết nhỏ hơn theo nghĩa dần dần sẽ tiến đến trường hợp suy biến).
- Tìm trường hợp suy biến.

Ví dụ:

1) Lập hàm $GT(n) = n!$

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

2) Dãy số *Fibonacci*: $F_1 = F_2 = 1;$
 $F_n = F_{n-1} + F_{n-2} \quad (n \geq 3)$

```
long F(n)
{ if (n ≤ 2) return 1;
  else return F(n-1)+F(n-2);
}
```

Nhận xét:

- Thông thường thay vì sử dụng lời giải đệ quy cho một bài toán, ta có thể thay thế bằng lời giải không đệ quy (khử đệ quy) bằng phương pháp lặp.
- Việc sử dụng thuật toán đệ quy có:

Ưu điểm	Khuyết điểm
Thuận lợi cho việc biểu diễn bài toán. Gọn (đối với chương trình).	Có khi không được tối ưu về thời gian. Có thể gây tốn bộ nhớ → xảy ra hiện tượng tràn bộ nhớ ngăn xếp (Stack) nếu dữ liệu lớn.

- Chính vì vậy, trong lập trình người ta cố tránh sử dụng thủ tục đệ quy nếu thấy không cần thiết.

♣ Bài tập:

- 1) Viết hàm lũy thừa `float lt(float x, int n)` cho ra giá trị x^n .

- 2) Viết chương trình nhập vào số nguyên rồi đảo ngược số đó lại (không được dùng phương pháp chuyển số thành xâu).
- 3) Viết chương trình cho phép sản sinh và hiển thị tất cả các số dạng nhị phân độ dài n (có gồm n chữ số).

Ví dụ 1: Viết thủ tục in xâu đảo ngược của xâu X.

Trước khi xây dựng hàm InNguoc thì ta xây dựng hàm tách chuỗi con từ chuỗi mẹ trước từ vị trí là *batdau* và lấy *soluong* ký tự.

```
char *copy(char *chuoi,int batdau,int soluong)
{ int i; char *tam;
  tam=(char *)malloc(100);
  for(i=(batdau-1);i<strlen(chuoi)&& i<(batdau-1+soluong);i++)
    tam[i-(batdau-1)]=chuoi[i];
  tam[i]=NULL;
  return tam;
}
```

Cách 1:

- Trường hợp chung: + In ký tự cuối của xâu X.
+ Đảo ngược phần còn lại.
- Trường hợp suy biến: Nếu xâu rỗng thì không làm gì hết.

```
void InNguoc(X){
  if (X[0] !='')
  {
    cout << X[strlen(X)-1];
    InNguoc(copy(X,0,strlen(x)-2);
  }
}
```

Cách 2:

- Trường hợp chung: + Đảo ngược xâu X đã bỏ ký tự đầu tiên.
+ In ký tự đầu tiên của X.
- Trường hợp suy biến: Nếu xâu rỗng thì không làm gì hết.

```
void Innguoc(X){
  if (X!="")
```

```

    {
        InNguoc(copy(X, 1, strlen(X)-2);
        cout << X[0];
    }
}

```

Ví dụ 2: Bài toán tháp Hà nội: Cho ba cọc A, B, C; có n đĩa khác nhau được xếp theo thứ tự nhỏ trên lớn dưới nằm trên cọc A. Yêu cầu: Chuyển chồng đĩa từ cọc A sang cọc C với điều kiện:

- Mỗi lần chỉ được chuyển một đĩa.
- Không có trường hợp đĩa lớn được đặt trên đĩa nhỏ.
- Có thể dùng cọc B làm cọc trung gian.

➤ Tham số hoá bài toán: HaNoi(n, A, B, C) //char A, B, C

Trong đó: n: Số đĩa.

A: Cọc nguồn cần chuyển đĩa đi.

B: Cọc trung gian.

C: Cọc đích để chuyển đĩa đến.

Chương trình chính như sau:

```

void main()
{
    cin >> n;
    A= 'A'; B= 'B'; C= 'C';
    HaNoi(3, A, B, C);
}

```

➤ Thuật toán đệ quy:

- Trường hợp suy biến:

Nếu $n = 1$ thì chuyển đĩa từ cọc A qua cọc C

- Trường hợp chung ($n \geq 2$):

Thử với $n=2$: + Chuyển đĩa thứ nhất từ A sang B.

+ Chuyển đĩa thứ 2 từ A sang C.

+ Chuyển đĩa thứ nhất từ B sang C.

→ Tổng quát: + Chuyển (n -1) đĩa từ A sang B (C làm trung gian).

+ Chuyển 1 đĩa từ A sang C (B: trung gian)

+ Chuyển (n -1) đĩa từ B sang C (A: trung gian).

Suy ra thuật toán đệ quy:

```

void HaNoi(n, A, B, C)

```

```

{
    if (n==1) cout << A << "→" << C;
    else
    {
        HaNoi(n -1, A, C, B);
        HaNoi(1, A, B, C);
        HaNoi(n -1, B, A, C);
    }
}

```

3.3. Thuật toán quay lui:

Ta có thể dùng kỹ thuật đệ quy để diễn tả thuật toán quay lui. Bài toán sử dụng thuật toán quay lui thường có dạng: Xác định một bộ gồm n thành phần: x_1, x_2, \dots, x_n thoả mãn điều kiện B nào đó.

Phương pháp của thuật toán quay lui:

- Giả sử ta đã xác định được $i-1$ thành phần: x_1, x_2, \dots, x_{i-1} . Để xác định thành phần x_i , ta duyệt tất cả các khả năng có thể có của nó.

Ví dụ: x_i có thể có giá trị từ 1 đến 8; gọi j là các giá trị có thể có của x_i , lúc đó ta dùng câu lệnh For như sau: For ($j=1; j<8; j++$) ...

- Bây giờ, với mỗi khả năng j ta luôn kiểm tra xem j có được chấp nhận không? (liệu bộ (x_1, x_2, \dots, x_i) hiện tại có thoả mãn điều kiện B hay không?)

➤ Như vậy, xảy ra 2 trường hợp:

- Nếu chấp nhận j :

- Xác định x_i theo j : $x_i=j$;
- Sau đó, nếu i còn nhỏ hơn n thì ta tiến hành xác định x_{i+1} .
- Ngược lại ($i = n$) thì ta được một lời giải.
- Kiểm tra j tiếp theo.

- Nếu tất cả các khả năng của j không có khả năng nào được chấp nhận thì quay lại bước trước để xác định lại x_{i-1} . (Cơ chế hoạt động trong bộ nhớ của thuật toán đệ quy giúp có thể thực hiện được điều này).

➤ Việc xác định x_i có thể mô tả qua thủ tục đệ quy sau:

```

void Try(i) //Thử xem  $x_i$  sẽ nhận giá trị nào
for (mỗi khả năng  $j$  của  $x_i$ )
{
    if <Chấp nhận>
    {

```

```

        <Xác định  $x_i$  theo  $j$ >; // Ví dụ:  $x[i]=j$ ;
        if ( $i==n$ ) <Ghi nhận một lời giải>;
            else Try( $i+1$ );
    }
}

```

♣ Bài tập:

- 1) Tìm tất cả các hoán vị của một mảng gồm có n phần tử.
- 2) Bài toán 8 con hậu: Hãy tìm cách đặt 8 quân hậu trên một bàn cờ vua sao cho không có quân hậu nào có thể ăn các quân hậu khác.

CHƯƠNG 4: MẢNG VÀ DANH SÁCH TUYẾN TÍNH

4.1. Mảng và cấu trúc lưu trữ của mảng:

- Mảng là cấu trúc dữ liệu đơn giản và thông dụng trong nhiều ngôn ngữ lập trình.
- Mảng là một tập có thứ tự gồm một số cố định các phần tử có cùng quy cách.

Ví dụ: Trong C, để khai báo một dãy số nguyên n phần tử: $a[0], a[1], \dots, a[n-1]$ (với $n \leq 100$), ta khai báo mảng a như sau:

`int a[100];`

Lúc này, việc truy xuất sẽ thông qua các phần tử của mảng, ký hiệu: $a[0], a[1], \dots, a[99]$.

- Ma trận là một mảng 2 chiều.

Ví dụ: `float B[100][100];`

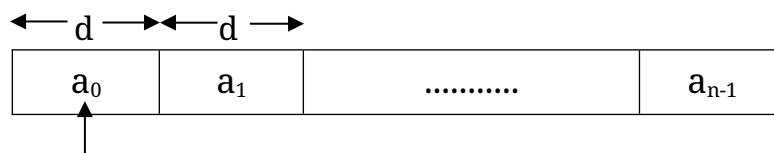
Khi đó, $B[i][j]$ là một phần tử của ma trận B. Trong đó i là hàng còn j là cột.

- Tương tự ta cũng có mảng 3 chiều, mảng 4 chiều.

❖ Cấu trúc lưu trữ:

Cách lưu trữ mảng thông thường (đối với mọi ngôn ngữ lập trình) là lưu trữ theo kiểu kế tiếp.

Ví dụ: Gọi a là mảng 1 chiều gồm có n phần tử, mỗi phần tử có độ dài là d (chiếm d byte) và được lưu trữ kế tiếp như hình dưới đây:



Loc (a_0): địa chỉ phần tử a_0

→ địa chỉ của phần tử thứ a_i :

$$\text{Loc} (a_i) = \text{Loc} (a_0) + d*i$$

Lưu ý:

- Đối với mảng nhiều chiều, việc tổ chức lưu trữ cũng được thực hiện tương tự:

Ví dụ: `int a[3][2];`

a[0][0]	a[0][1]	a[1][0]	a[1][1]	a[2][0]	a[2][1]
---------	---------	---------	---------	---------	---------

→ Địa chỉ của phần tử a_{ij} :

$$\text{Loc}(a[i][j]) = \text{Loc}(a[0][0]) + d \cdot (i \cdot n + j)$$

Trong đó, n là số cột của ma trận.

♣ Bài tập:

1) Viết công thức tổng quát để tính địa chỉ của một phần tử nào đó của một mảng n chiều ($\text{Loc } a[i_0, \dots, i_{n-1}]$), với chỉ số các chiều này lần lượt là: $b_0..b'_0, b_1..b'_1, \dots, b_{n-1}..b'_{n-1}$; trong đó: $i_0 \in [b_0..b'_0], i_1 \in [b_1..b'_1], \dots, i_{n-1} \in [b_{n-1}..b'_{n-1}]$. Địa chỉ này phụ thuộc vào địa chỉ của chỉ số đầu tiên $a[b_0, b_1, \dots, b_{n-1}]$. Cho d là độ dài của một phần tử.

Lưu ý: do các phần tử của mảng thường được lưu trữ kế tiếp nhau nên việc truy nhập vào chúng nhanh, đồng đều với mọi phần tử (ưu điểm). Trong lúc đó, nhược điểm của việc lưu trữ mảng là:

- + Phải khai báo chỉ số tối đa, do đó có trường hợp gây lãng phí bộ nhớ.

- + Khó khăn trong việc thực hiện phép xoá / chèn một phần tử trong mảng.

2) Giả sử trong bộ nhớ có mảng a gồm n phần tử a_0, a_1, \dots, a_{n-1} .

Hãy viết các hàm sau:

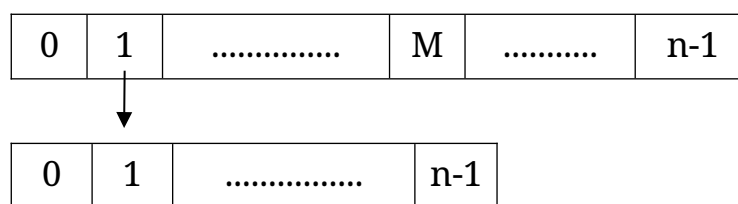
- + `void Xoa(i)`: Xoá phần tử thứ i trong mảng này.
- + `void ChenSau(i, x)`: Chèn sau phần tử thứ i một phần tử có giá trị là x .

4.2. Danh sách tuyến tính (Linear list):

❖ Định nghĩa:

Danh sách tuyến tính là một dãy có thứ tự a_1, a_2, \dots, a_n ($n \geq 0$). Nếu $n=0$ được gọi là danh sách rỗng. Ngược lại: a_1 được gọi là phần tử đầu tiên, a_n được gọi là phần tử cuối cùng, và n được gọi là chiều dài của danh sách.

- Đối với danh sách tuyến tính, với mỗi phần tử a_i ($i = 1, n-1$) thì có phần tử tiếp theo là a_{i+1} và với mỗi phần tử a_i ($i = 2..n$) thì có phần tử đứng trước là a_{i-1} .
- Danh sách tuyến tính khác cơ bản với mảng một chiều ở chỗ là kích thước của danh sách không cố định bởi vì phép bổ sung và phép loại bỏ thường xuyên tác động lên một danh sách. Ví dụ: Stack.
- Có nhiều cách để lưu trữ một danh sách tuyến tính:
 - + Lưu trữ theo địa chỉ kế tiếp bằng mảng 1 chiều.
 - + Lưu trữ địa chỉ bằng con trỏ (sử dụng danh sách móc nối).
 - + Lưu trữ ra file (sử dụng bộ nhớ ngoài).
- Với danh sách tuyến tính, ngoài phép bổ sung và loại bỏ còn có một số phép sau:
 - + Phép ghép 2 hoặc nhiều danh sách thành một danh sách (xem như bài tập, làm trên mảng và trỏ).



- + Phép tách (tách một danh sách thành 2 danh sách).
- + Sao chép một danh sách ra nhiều danh sách (2 danh sách).
- + Cập nhật hoặc sửa đổi nội dung các phần tử của danh sách.
- + Sắp xếp các phần tử trong danh sách theo thứ tự ấn định trước.
- + Tìm kiếm một phần tử trong danh sách thoả mãn một điều kiện cho trước.

4.3. Ngăn xếp (Stack):

4.3.1. Định nghĩa:

Stack là một kiểu danh sách tuyến tính đặc biệt, trong đó phép bổ sung và loại bỏ chỉ thực hiện ở một đầu gọi là đỉnh Stack (đầu kia gọi là đáy của Stack).

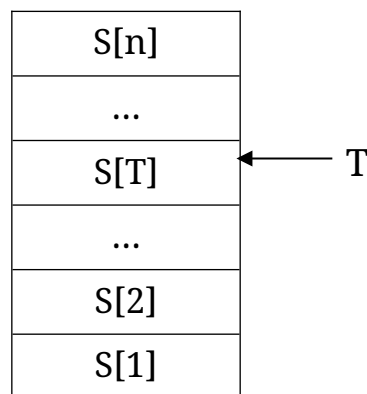
Nguyên tắc bổ sung và loại bỏ đối với Stack được gọi là nguyên tắc vào sau ra trước (LIFO – Last In First Out).

4.3.2. Lưu trữ Stack bằng mảng:

Vì Stack là một danh sách tuyến tính nên có thể sử dụng mảng một chiều để tổ chức một Stack. Chẳng hạn: sử dụng mảng S để lưu trữ dãy các phần tử: S[1], S[2],..., S[n] (n gọi là số phần tử cực đại của mảng S).

Gọi T là chỉ số của phần tử đỉnh của Stack. T được sử dụng để theo dõi vị trí đỉnh của Stack nên nếu sử dụng danh sách móc nối để tổ chức một Stack thì T được xem như là một con trỏ chỉ vào vị trí đỉnh của Stack.

Giá trị của T sẽ tăng lên một đơn vị khi bổ sung một phần tử vào danh sách và sẽ giảm bớt 1 khi loại bỏ một phần tử ra khỏi Stack.



Lưu ý:

- Khi $T = n$ thì không thể bổ sung thêm (hay nói cách khác là Stack đầy).
- Khi $T = 0$ thì không thể loại bỏ phần tử vì khi đó Stack rỗng (hay Stack cạn).

➤ Thuật toán bổ sung một phần tử X vào Stack S có đỉnh là T:

```
void Push(S, &T, X)    //T: tham biến
    if (T==n) cout << "Không bổ sung được";
    else
    {
        T=T+1;
        S[T]=X;
    }
    return;
```

➤ Thuật toán loại bỏ khỏi Stack S phần tử tại đỉnh T và gán giá trị của phần tử đó cho tham biến X:

```
void Pop(S, &T, &X)    // T, X: tham biến
    if (T==0) cout << "Stack cạn"
    else
```



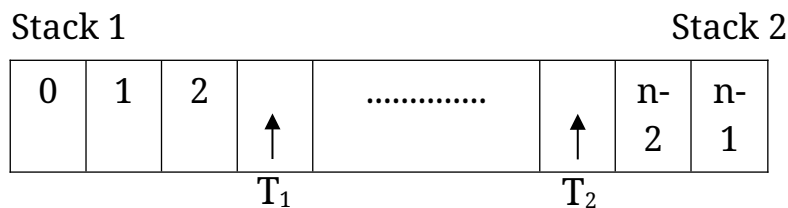
```

    {
        X= S[T];
        T=T-1;
    }
    return;

```

♣ Bài tập:

Giả sử ta có mảng S dùng để lưu trữ đồng thời hai Stack như hình dưới đây:



Trong đó:

- + T_1 dùng để theo dõi vị trí đỉnh của Stack 1 (đây là phần tử số 1 của mảng S).
- + T_2 dùng để theo dõi vị trí đỉnh của Stack 2 (đây là phần tử n của mảng S).

=> Nếu $T_2 = T_1 + 1$ thì ngưỡng, lúc này cả hai Stack cùng đầy.

Viết các hàm sau:

void Bosung(i, x)

Dùng để bổ sung vào Stack i ($i = \overline{1, 2}$) một phần tử x.

2) void LoaiBo (i, x)

Dùng để loại bỏ 1 phần tử ra khỏi Stack i ($i = \overline{1, 2}$) và trả về phần tử này cho tham biến x.

4.3.3. Các ví dụ:

Ví dụ 1: Viết chương trình đổi một số hệ thập phân thành số hệ nhị phân.

void Chuyen

1. cin >> n; T=0;

2.while (n!=0)

{

 r= n % 2;

 Push(S, T, r);

```
        n= n / 2;
    }
3. while (T!=0)
    {
        Pop(S, T, r);
        cout << r;
    }
return; // Lưu ý: int S[100]
```

Ví dụ 2: Viết chương trình tính giá trị của một biểu thức hậu tố (tức là ký pháp Ba Lan đảo).

➤ *Một số khái niệm:*

- Biểu thức số học mà ta thường sử dụng được gọi là biểu thức trung tố. Ở đây, ta coi các thành phần (token) có trong một biểu thức trung tố bao gồm: hằng số (toán hạng), toán tử (+, -, *, /), các dấu ngoặc: (,).
- Biểu thức số học còn có thể biểu diễn dưới dạng ký pháp hậu tố (biểu thức hậu tố) và ký pháp tiền tố (biểu thức tiền tố).

Ví dụ: $2 + 3 \rightarrow$ biểu thức trung tố.

$2\ 3\ + \rightarrow$ biểu thức hậu tố (các toán tử đi sau các toán hạng).

$+ 2\ 3 \rightarrow$ biểu thức tiền tố (các toán tử đi trước các toán hạng).

- Với các ký pháp mới này (ký pháp Ba Lan), dấu ngoặc là không cần thiết.

Ví dụ: $7 + 3 * 5 \rightarrow 7\ 3\ 5\ *\ +$

$7 * 3 + 5 \rightarrow 7\ 3\ *\ 5\ +$

$7 * (3 + 5) \rightarrow 7\ 3\ 5\ +\ *$

$(1 + 5) * (8 - (4 - 1)) \rightarrow 1\ 5\ +\ 8\ 4\ 1\ -\ -\ *$

- Cách tính giá trị một biểu thức hậu tố: Biểu thức cần tính được đọc từ trái sang phải cho tới khi tìm ra 1 toán tử. Hai toán hạng được đọc gần nhất (trước toán tử này) sẽ được thực hiện với nhau bởi toán tử đó để thay bằng một toán hạng mới. Quá trình này lại tiếp tục cho đến khi có được kết quả cuối cùng.

➤ *Thuật toán:*

1) $T = \emptyset$;

2) do

 Đọc token X tiếp theo trong biểu thức;

 if <X là số> Push(S, T, X);

 if <X là toán tử>

 {

 Pop(S, T, Y);

 Pop(S, T, Z);

 Tác động toán tử X vào Z và Y rồi gán kết quả cho W

 Push(S, T, W);

 }

 while (<vẫn còn token>);

3) Pop(S, T, X);

4) cout << X;

♣ Bài tập:

- 1) Chuyển thuật toán trên thành chương trình C.
- 2) Nhập chuỗi có nội dung là biểu thức hậu tố, các token cách nhau 1 ô trống. Viết chương trình tính kết quả của biểu thức vừa nhập.

Ví dụ 3: Viết chương trình để chuyển biểu thức trung tố sang hậu tố.

➤ *Thuật toán:*

- 1) Khởi tạo Stack chứa các ký tự toán tử có đỉnh là T:

T=0; { char s[100] chứa các ký tự → +, -, *, /, (}

Khởi tạo chuỗi biểu thức hậu tố:

Xau="";

- 2) do

<Đọc một token ở biểu thức trung tố từ trái sang phải;>

switch (Token)

{

case *<Token là toán hạng>*:

{ Cộng vào bên phải của chuỗi (kèm khoảng trắng). }

Xau=Xau + Token + " ";break;

case *<Token là toán tử>*:

if (*<Stack rỗng>*) *<Push Token này>*;

else

{

// So sánh token này với toán tử ở đỉnh Stack;

if (<Token > toán tử ở đỉnh Stack>

<Push Token này>;

else

{

- <Lặp việc Pop các toán tử ở đỉnh Stack cộng vào bên phải của chuỗi (trừ dấu " (") cho tới khi Token > toán tử ở đỉnh hoặc Stack rỗng>;

- <Push Token này>;

}

} break;

case *<Token là "(">*: *<Push Token này>;break;*

case *<Token là ">*:

<Lắp việc Pop các toán tử ở đỉnh Stack cộng vào bên phải của xâu (trừ dấu ngoặc mở) cho tới khi gặp dấu ngoặc mở.>; break;

}

while (<chưa hết chuỗi trong biểu thức trung tố>);

3) Pop các phần tử còn lại trong Stack vào bên phải của xâu cho đến khi Stack rỗng rồi đưa vào bên phải xâu.

4) cout << Xâu;

♣ Bài tập:

Nhập một xâu có nội dung là biểu thức trung tố. Tính kết quả biểu thức này.

Chú ý: Các toán tử: *, /, +, -, (,). Dùng hàm trả về thứ tự ưu tiên để so sánh:

* hay / → trả về 2.

+ hay - → trả về 1.

(hay) → trả về 0.

4.3.4. Stack với việc cài đặt thuật toán đệ quy:

Việc cài đặt một thuật toán đệ quy được tổ chức trong bộ nhớ dưới dạng Stack. Cụ thể: Khi một chương trình con đệ quy được gọi từ chương trình chính thì ta nói chương trình con được thực hiện ở mức 1. Và trong chương trình con, gặp lời gọi của chính nó thì chương trình con lần lượt được thực hiện ở các mức 2, mức 3, ..., mức k (mức k phải được hoàn thành xong thì mức k-1 mới được thực hiện tiếp).

Khi từ mức i đi sâu vào mức i+1 thì có thể có một số tham số, biến cục bộ, địa chỉ quay lui ứng với mức i sẽ được bảo lưu để khi quay về chúng sẽ được khôi phục để tiếp tục sử dụng.

Những tham số của biến cục bộ, những địa chỉ quay lui được bảo lưu sau thì nó lại được khôi phục trước.

Sử dụng Stack trong việc cài đặt chương trình con đệ quy theo hình thức sau:

- Khi có lời gọi đến chính nó thì Stack sẽ được bổ sung một phần tử (là một record gồm các trường: tham số, biến cục bộ, địa chỉ quay lui).

- Khi thoát khỏi một mức thì 1 phần tử ở đỉnh Stack sẽ được lấy ra (khôi phục lại giá trị cần thiết trước đây).

➤ *Ta có thể tóm tắt các bước này như sau:*

Bước 1: Bước mở đầu (bản chất là Push): Bảo lưu tham số, biến cục bộ và địa chỉ quay lui.

Bước 2: Bước thân.

Chia làm 2 trường hợp:

- Nếu gặp trường hợp suy biến thì thực hiện phần kết thúc. Chuyển tới bước 3.
- Nếu ngược lại thì thực hiện phần tính từng phần và chuyển sang bước 1.

Bước 3: Bước kết thúc. Khôi phục lại tham số, biến cục bộ và địa chỉ quay lui (pop). Và chuyển đến địa chỉ quay lui này.

Chú ý: Dựa vào nguyên tắc này mà Stack thường được sử dụng để biến đổi một thuật toán đệ quy thành một thuật toán không đệ quy.

Ví dụ 1: Bài toán tháp Hà Nội:

```
struct Rec
{
    int nn;
    char aa, bb, cc;
};

int T;
char a, b, c;
Rec r;
rec S[100];
Rec rr;

void BoVao(rec r;int n;char a;char b;char c)
{
    r.nn= n; r.aa=a; r.bb=b; r.cc=c;
    T=T+1; S[T]=r;
}

void LayRa(Rec r)
```

```

{
    r=S[T];
    T=T-1;
}

void main()
{
    cin >> n;
    a='A'; b='B'; c='C';
    T=0;
    BoVao(r, n, a, b, c);
    do
    {
        LayRa(rr);
        if (rr.nn==1) cout << rr.aa << "→" << rr.cc;
        else
        {
            BoVao(r, rr.nn-1, rr.bb, rr.aa, rr.cc);
            BoVao(r, 1, rr.aa, rr.bb, rr.cc);
            BoVao(r, rr.nn-1, rr.aa, rr.cc, rr.bb);
        }
    }
    while (T!=0);
}

```

Ví dụ 2:

Chương trình sau mô tả thuật toán tính $n!$ theo kiểu đệ quy nhưng bằng cách sử dụng Stack: là mảng mà mỗi phần tử của nó là một record gồm 2 trường: trường Para (tham số) và trường Addr (địa chỉ quay lui).

```

struct Rec {
    int Para;
    int Addr;
};

int n, T;
Rec a[100];
float Kq;
Rec TempRec;

void Push(Rec TempRec){

```

```

        T=T+1;
        a[T]=TempRec;
    }

    void Pop(Rec &TempRec){

        TempRec=a[T];
        T=T-1;
    }

    void main()
    {
        cin >> n;
        T=0;
        TempRec.Para=n;
        TempRec.Addr=5;
        1: Push(TempRec);
        2: if (a[T].Para==0)
            {
                Kq=1;
                goto 4;
            }
        else
            {
                TempRec.Para=a[T].Para-1;
                TempRec.Addr=3;
            }
        goto 1;
        3: Kq=Kq*a[T].Para;
        4: Pop(TempRec);
        switch (TempRec.Addr){
            case 3: goto 3;break;
            case 5: goto 5;break;
        }
        5: cout << Kq;
    }

```


4.4. Hàng đợi (Queue):

4.4.1. Định nghĩa:

Hàng đợi là một danh sách tuyến tính mà phép bổ sung được thực hiện ở một đầu (gọi là lối vào/lối sau: rear) và phép loại bỏ được thực hiện ở đầu kia (lối ra/lối trước: front).

Nhận xét: Cơ cấu của Queue giống như một hàng đợi: vào trước - ra trước. do đó Queue còn được gọi là danh sách kiểu FIFO (First In First Out).

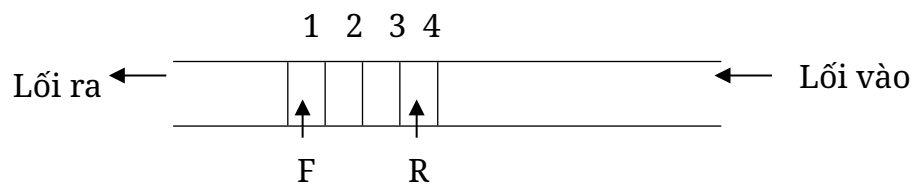
4.4.2. Lưu trữ Queue bằng mảng:

Có thể dùng mảng một chiều Q có n phần tử làm cấu trúc lưu trữ của hàng đợi. Để xử lý Q ta dùng 2 biến:

- + Biến R để theo dõi vị trí lối sau của Q.
- + Biến F để theo dõi vị trí lối trước của Q.

Lưu ý:

- Khi Q (Queue) rỗng thì ta quy ước: $F = R = 0$.
- Khi một phần tử được bổ sung thì R tăng lên 1 ($R=R+1$). Khi lấy bớt một phần tử ra thì F tăng lên 1 ($F=F+1$).



Tuy nhiên, với cách tổ chức này có thể xuất hiện tình huống là đến một lúc nào đó không thể bổ sung tiếp phần tử nào nhưng không gian nhớ của mảng Q vẫn còn chỗ. Để khắc phục, ta xem Q như là một mảng vòng tròn, nghĩa là xem $Q[1]$ đứng sau $Q[n]$.

Với cách tổ chức này ta có:

➤ *Thuật toán bổ sung vào hàng đợi Queue (có vị trí lối trước là F và vị trí lối sau là R) phần tử x:*

```
void Insert_Queue(&Q, &F, &R, &X) //Q, R, F: tham biến
{
    if ((R % n) + 1 == F)
        // Tương đương: if ((R < n) && (R + 1 == F)) || ((R == n) && (F == 1))
        cout << "Hàng đợi đã đầy!"
    else
    {
        R = (R % n) + 1;
        Q[R] = X;
    }
}
```

```

        if (F==0) F=1;
    }
return;

```

➤ Thuật toán loại bỏ một phần tử từ hàng đợi Queue (lỗi trước F, lỗi sau là R) và phần tử loại bỏ được gán cho một biến X:

```

void Delete_Queue(&Q, &F, &R, &X) //Q, F, R, X: tham biến
    if (F==0) printf("Hàng đợi đang cạn!");
    else
    {
        X=Q[F];
        if (F==R) F=R=0;
        else F=(F % n)+1;
    }
return;

```

♣ Bài tập:

1) Tính giá trị của một biểu thức trung tố mà các token có 2 loại (Hằng số, toán tử: +, -, *, /) bằng phương pháp sau:

- Đọc các token từ trái sang phải, tất cả đưa vào hàng đợi. Ví dụ: $11 + 2 * 3$:

	11	+	2	*	3	
--	----	---	---	---	---	--

- Lần lượt lấy ra để bỏ vào một danh sách thứ hai. Nhớ rằng: nếu gặp phải toán tử * hoặc / thì lấy ở hàng đợi một phần tử và ở danh sách thứ 2 lấy ra lại một phần tử để thực hiện phép toán này, được kết quả lại bỏ vào danh sách thứ 2.

	11	+	2	
--	----	---	---	--

$2 * 3$

2) Giống như bài tập 1, nhưng các token có thể có dấu '(' hoặc dấu ')', bằng phương pháp sau:

Ví dụ: $1 + (2 * (3 + 4))$

1	(2	*	(3	+	4		
---	---	---	---	---	---	---	---	--	--

- Lần lượt đọc các token từ trái sang phải để push vào một Stack cho đến khi gặp dấu ')' thì lần lượt lấy các phần tử ở trong Stack này để bỏ vào một danh sách thứ hai (bỏ vào từ phía trái) cho đến khi gặp dấu '('.

	3	+	4	
--	---	---	---	--

- Lúc đó ta xử lý danh sách thứ 2 này (tính) dựa vào thủ tục đã xây dựng trong bài tập 1). Được kết quả lại cho vào Stack ban đầu.

1	+	(2	*	7	
---	---	---	---	---	---	--

- Rồi lại tiếp tục cho đến khi hết biểu thức này.

1	+	14	
---	---	----	--

- Lúc đó ta coi Stack này như một hàng đợi để sử dụng thủ tục trong bài tập 1 mà xử lý.

Nhận xét: Một Stack cũng có thể xem như là một Queue hoặc là một danh sách tuyến tính nói chung. Vấn đề quan trọng là ta cần sử dụng 2 biến để theo dõi vị trí 2 đầu của danh sách này để có thể thực hiện phép bổ sung hay loại bỏ cho phù hợp.

CHƯƠNG 5: DANH SÁCH MÓC NỐI (LINKED LIST)

5.1. Danh sách móc nối đơn:

5.1.1. Tổ chức danh sách móc nối đơn:

- Mỗi phần tử của danh sách được gọi là nút (node), là một bản ghi gồm 2 phần:
 - Phần thông tin (Info): Chứa thông tin của phần tử (có thể có nhiều hơn một trường).
 - Phần liên kết (Next): Đây là một trường chứa địa chỉ của phần tử ngay sau nó (là một con trỏ). Trường này có kiểu dữ liệu con trỏ.
- Các nút có thể nằm rải rác trong bộ nhớ.
- Để có thể truy cập đến mọi phần tử của danh sách, ta phải truy nhập vào nút đầu tiên. do đó phải có con trỏ First để trỏ vào phần tử đầu tiên của danh sách. Từ nút đầu tiên, thông qua trường Next ta sẽ đi đến nút thứ hai và cứ như thế ta có thể duyệt hết các phần tử trong danh sách.
- Phần tử cuối cùng trong danh sách có trường Next không chứa địa chỉ của phần tử nào cả mà ta gọi là NULL.
- Khi danh sách rỗng, ta quy ước First = NULL;
- Ta ký hiệu:
 - **p = new <kiểu>;** là thủ tục nhằm tạo một vùng nhớ còn trống để chứa một nút và nút này được trỏ bởi con trỏ p (p chứa địa chỉ nút này).
 - **delete p;** là thủ tục để giải phóng vùng nhớ của nút trỏ bởi con trỏ p khỏi bộ nhớ.
- Sử dụng ký hiệu -> để truy cập đến các biến là các trường có trong một nút được trỏ bởi p.

Ví dụ:

```
struct Nut
{
    int      Info;
    Nut*     Next;
```

```
};
Nut* First;
Nut* p;
```

5.1.2. Một số phép toán trên danh sách nối đơn:

5.1.2.1. Chèn một nút mới có nội dung X vào danh sách sau nút được trỏ bởi p:

```
void Insert_Node(&First, p, X); //First: tham biến
    Tam= new Nut;
    Tam->Info=X;
    if (First==NULL)
    {
        First=Tam;
        First->Next=NULL;
        return;// thoát khỏi chương trình con
    }
    Tam->Next=p->Next;
    P->Next=Tam;
return;
```

5.1.2.2. Loại bỏ một nút đang trỏ bởi p ra khỏi danh sách:

```
void Delete_Node(&First, p); //First: tham biến
    if (First==NULL)
    {
        cout << "Danh sách rỗng";
        return;
    }
    if (First==p)
    {
        First=First->Next;
        delete p;
        return;
    }
    q=First;
    while (q->Next!=p) q=q->Next;
    q->Next=p->Next;
    delete p;
return;
```

5.1.2.3. Ghép 2 danh sách được trỏ bởi first1 và first2 thành một danh sách được trỏ bởi first1:

```
void Combine(&First1, First2); //First1: tham biến
```

```

if (First1==NULL)
{
    First1=First2;
    return;
}
if (First2==NULL) return;
p=First1;
while (p->Next!=NULL) p=p->Next;
p->Next=First2;
return;

```

♣ Bài tập:

Tạo file văn bản tên là VB.TXT có cấu trúc như sau:

Viết thủ tục:

- 1) void docfile(Nut **first; FILE *f); để lần lượt đọc các dòng trong file VB.TXT và đưa ra một danh sách nối đơn có phần tử đầu trỏ bởi first, kiểu dữ liệu là con trỏ như khai báo trước (ở ví dụ).

Tên(6 ký tự)	
	Tuổi
Lan	25
Le	20

Gợi ý:

```

F=fopen("VB.TXT","rt");
*First=NULL;
while Eof(f) do
{
    fgets(Xau,6,f);
    fscanf(f,"%d",&So);
    Tam=new Nut;
    Tam->Name=Xau;
    Tam->Age=So;
    Tam->Next=First;
    First=Tam;
}

```

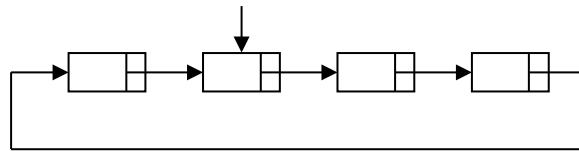
- 2) void Dinhvi(p, i) //p: tham biến
Để định vị con trỏ p đến phần tử thứ i trong danh sách.
- 3) void Lietke(first)
Để liệt kê nội dung của các nút trong danh sách.

5.2. Danh sách nối vòng:

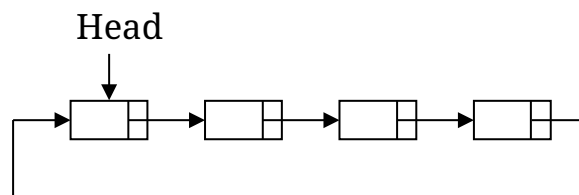
5.2.1. Nguyên tắc:

Trong danh sách nối đơn, trường Next của nút cuối danh sách có giá trị NULL, để tạo nên sự linh hoạt trong việc truy cập đến các phần tử của danh sách, người ta cho trường Next của nút này lại trở đến nút đầu của danh sách và được danh sách có cấu trúc như sau:

T

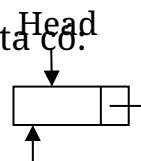


Trong danh sách này có một con trỏ T chạy. Trong trường hợp nối đơn thì đứng ở mỗi nút ta chỉ có thể truy cập đến các phần tử đứng sau nó nhưng với danh sách nối vòng, ta có thể truy cập vào tất cả các nút của danh sách từ bất kỳ nút nào. Song cách tổ chức này có thể dẫn đến tình trạng là truy cập không kết thúc. Nhược điểm này có thể được khắc phục bằng cách thêm một nút vào danh sách gọi là nút đầu danh sách và biến trỏ Head chứa địa chỉ của nút đầu này.



Nội dung của trường Info của nút này (trỏ bởi Head) không chứa thông tin nào.

Trong trường hợp danh sách rỗng, ta có: Head-→Next = Head



5.2.2. Thuật toán bổ sung và loại bỏ một nút của danh sách nối vòng:

5.2.2.1. Bổ sung một nút có nội dung trường Info là X vào ngay sau nút đầu danh sách Head:

```
void Insert_Node(Head, X) // Head: tham trị
    Tam=new Nut;
    Tam->Info=X;
    Tam->Next=Head->Next;
    Head->Next=Tam;
```

```
return,
```

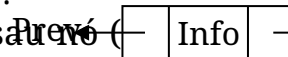
5.2.2.2. Loại bỏ một nút đứng ngay sau Head:

```
void Delete_Node(Head)
{
    if ( Head->Next==Head)
    {
        cout << "Danh sách rỗng";
        return;
    }
    Tam=Head->Next;
    Head->Next=Tam->Next;
    delete Tam;
    return;
}
```

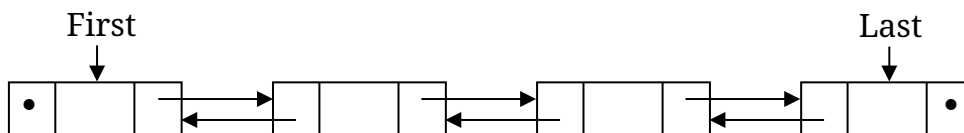
5.3. Danh sách nối kép:

5.3.1. Tổ chức:

Tương tự như danh sách nối đơn hoặc nối vòng, danh sách nối kép bao gồm các nút có thể nằm rải rác trong bộ nhớ và được liên kết với nhau. Nhưng điều khác biệt ở đây là tại mỗi nút có hai trường chứa địa chỉ của nút đứng trước và nút đứng sau nó. Như hình vẽ minh họa, mỗi nút có dạng:



Lúc đó danh sách có dạng như sau:

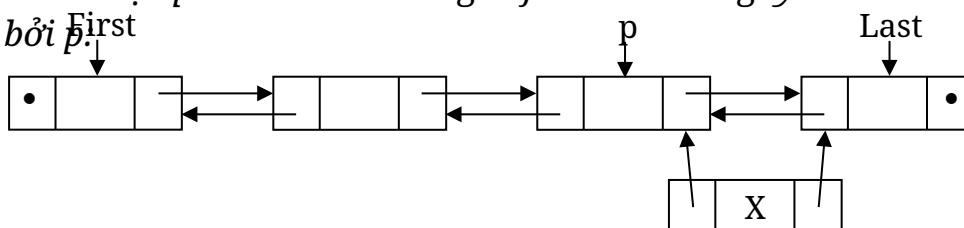


Nhận xét:

- Danh sách này sử dụng 2 biến con trỏ First và Last để trỏ tới nút đầu tiên và nút cuối cùng. Trong đó trường Prev của nút đầu tiên và trường Next của nút cuối cùng có giá trị là NULL.
- Khi danh sách rỗng: First = Last = NULL.

5.3.2. Một số phép toán trên danh sách nối kép:

➤ Chèn một phần tử có trường Info là X vào ngay sau nút được trỏ bởi p.




```

void Insert_Node(&First,&Last,p,X)//First,Last:tham biến
    Tam=new Nut;
    Tam->Info=X;
    if (First==NULL)
    {
        First=Tam;
        First->Next=First->Prev=NULL,
        Last=First;
        return;
    }
    Tam->Next=p->Next;
    Tam->Prev=p;
    p->Next=Tam;
    if (p!=Last)
        Tam->Next->Prev=Tam;
    else Last=Tam;
return;

```

➤ *Loại bỏ một nút trở bởi p ra khỏi danh sách:*

```

void Delete_Node(First, Last, p)//First,Last:tham biến
    if (First==NULL)
    {
        cout << "Danh sách rỗng";
        return;
    }
    if (First==Last) First=Last=NULL;
    else if (p==First)
    {
        First=First->Next;
        First->Prev=NULL;
    }
    else if (p==last)
    {
        Last=Last->Prev;
        Last->Next=NULL;
    }
    else
    {
        P->Prev->Next=p->Next;
        P->Next->Prev=p->Prev;
    }
}

```

```

delete p;
return;

```

5.4. Ví dụ về việc sử dụng danh sách móc nối:

- Biểu diễn một đa thức bằng một danh sách móc nối đơn.
- Đa thức sẽ được biểu diễn dưới một danh sách móc nối đơn mà mỗi nút (lưu một đơn thức) có dạng như sau:

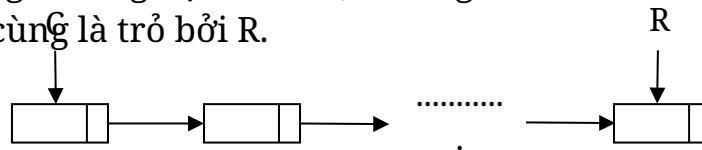
Hệ số	Mũ	Tiếp
-------	----	------

Bài toán: Tính tổng 2 đa thức:

- Giả sử đa thức $A(x)$, $B(x)$ sẽ được biểu diễn bởi 2 danh sách móc nối đơn lần lượt trở bởi A và B.
- Vấn đề đặt ra: Tạo danh sách móc nối đơn thứ 3 trở bởi C để biểu diễn cho đa thức:

$$C(x) = A(x) + B(x)$$

- Trước hết ta viết thủ tục để ghép thêm một nút vào cuối danh sách C có nội dung trường hệ số là XX, trường mũ là YY. Giả sử danh sách C có nút cuối cùng là trở bởi R.



```

void Ghep(C, R, XX, YY)//C:tham trị,R:tham biến

```

```

1. Tam=new Nut;
   Tam->Heso=XX;
   Tam->Mu=YY;
   Tam->Tiep=NULL;
2. R->Tiep=Tam;
   R=Tam;
return;

```

```

void Cong_Da_Thuc(A, B, C)//A,B:tham trị,C:tham biến

```

```

1. C=new Nut;
   R=C;
   p=A; q=B;
2. while ((p!=NULL) && (q!=NULL))
   if (p->Mu==q->Mu)
   {
       XX=p->Heso+q->Heso;
       if (XX!=0) Ghep(C, R, XX, p->Mu);
   }

```

```

        p=p->Tiep;
        q=q->Tiep;
    }
    else if (p->Mu < q->Mu)
    {
        Ghep(C, R, q->Heso, q->Mu);
        q=q->Tiep;
    }
    else
    {
        Ghep(C, R, p->Heso, p->Mu);
        p=p->Tiep;
    }
3. while (q!=NULL)
    {
        Ghep(C, R, q->Heso, q->Mu);
        q=q->Tiep;
    }

    while (p!=NULL)
    {
        Ghep(C, R, p->Heso, p->Mu);
        p=p->Tiep;
    }
    Tam=C;
    C=C->Tiep;
    delete Tam;
return;

```

5.5. Stack và Queue móc nối:

Đối với Stack, việc truy nhập luôn thực hiện ở một đầu nên việc cài đặt một danh sách bằng Stack móc nối là khá tự nhiên. Chẳng hạn với danh sách nối đơn có nút đầu trỏ bởi First thì có thể coi First như là đỉnh Stack.

Bổ sung một phần tử vào Stack cũng chính là bổ sung một nút vào danh sách để nút đó trở thành nút đầu tiên trong danh sách. Loại bỏ một phần tử của danh sách chính là loại bỏ phần tử đầu tiên. Đối với danh sách móc nối, chúng ta không cần kiểm tra hiện tượng tràn Stack vì Stack dùng danh sách móc nối không bị giới hạn kích thước như dùng mảng (mà chỉ giới hạn bởi bộ nhớ toàn phần).

- *Thủ tục chèn vào đầu danh sách một phần tử:*

```
void Push(&First, X)//First:tham biến
    p=new Nut;
    p->Info=X;
    p->Tiep=First;
    First=p;
return;
```

- *Thủ tục lấy phần tử ở đầu danh sách:*

```
void Pop(&First, &X)//First:tham biến
    if (First==NULL)
    {
        cout << "Stack cạn";
        return;
    }
    X=First->Info;
    p=First;
    First=First->Tiep;
    delete p;
return;
```

Đối với Queue, dùng danh sách móc nối cũng theo phương pháp tương tự. Nhưng nếu dùng danh sách móc nối đơn thì cần 2 biến con trỏ First và Last.

Bổ sung một phần tử vào Queue là bổ sung một phần tử ngay sau Last. Và loại bỏ một phần tử khỏi Queue là loại bỏ phần tử đầu tiên (First).

- *Thủ tục chèn vào đầu danh sách một phần tử:*

```
void Insert_Queue(&First, &Last, X) //First,Last:tham biến
    p=new Nut;
    p->Info=X;
    p->Tiep=NULL;
    if (Last==NULL) First=Last=p;
    else
    {
        Last->Next=p;
        Last=p;
    }
return;
```

- *Hàm xoá phần tử trên Queue:* giống như thủ tục Pop ở trên.

♣ Bài tập (Bài thực hành số 2):

Tạo từ điển: Có một menu làm các công việc sau:

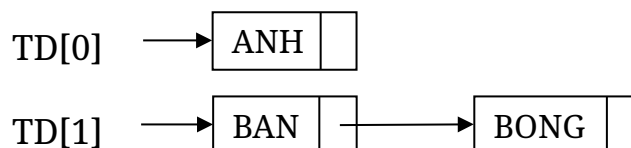
1) Khởi tạo từ điển TD từ file TD.DAT:

Đọc file TD.DAT là văn bản (file này đã chứa sẵn các từ có tối đa là 7 ký tự). Mỗi từ chỉ có các ký tự trong tập ['A'...'Z']. Các từ trong file đã được sắp xếp theo thứ tự ABC) và lưu các từ này vào một mảng các danh sách móc nối. Cụ thể: Nut * TD[26]; //lưu 26 danh sách tương ứng chữ cái['A'..'Z']

Trong đó, kiểu Nut được khai báo như sau:

```
Struct Nut
{
    char Tu[7];
    Nut *Tiep;
}
```

Ví dụ: File TD.DAT có 3 từ: ANH, BAN, BONG.



Còn TD[2],..., TD[25] đều là NULL.

Lưu ý: Nếu file này chưa có thì cho các phần tử của mảng đều là NULL.

2) Liệt kê tất cả các từ trong TD:

Gõ vào ký tự, sau đó hiển thị tất cả các từ có ký tự đầu là ký tự được gõ vào.

3) Bổ sung một từ:

Từ bàn phím gõ vào một từ. Nếu từ đó chưa có trong TD thì bổ sung nó vào TD, còn ngược lại thì thông báo: “Từ này đã có trong TD”.

4) Xoá một từ:

Từ bàn phím gõ vào một từ. Nếu từ đó có trong TD thì xoá khỏi TD, còn ngược lại thì thông báo: “Không có từ này trong TD”.

5) Cập nhật từ điển từ một file văn bản:

Đọc một file văn bản bất kỳ, trong đó có chứa các từ (một từ được quy định là các ký tự liên tiếp trong tập ['A'...'Z']) các ký tự còn lại đều

coi là dấu phân cách). Cứ mỗi từ đọc được trong file văn bản này hãy thực hiện công việc sau: Nếu từ đó không tìm thấy trong TD thì chèn nó vào vị trí thích hợp.

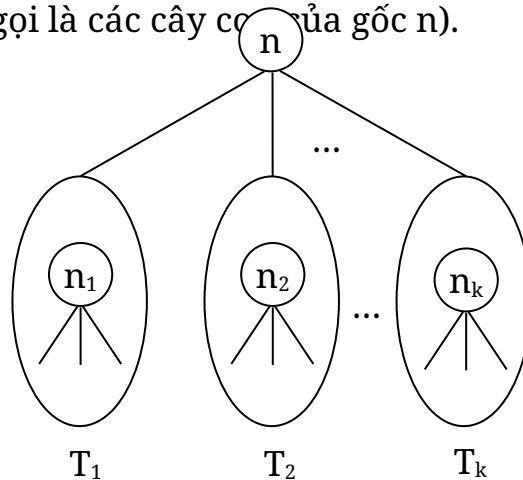
6) *Lưu TD vào file **TD.DAT**.*

CHƯƠNG 6: CÂY (TREE)

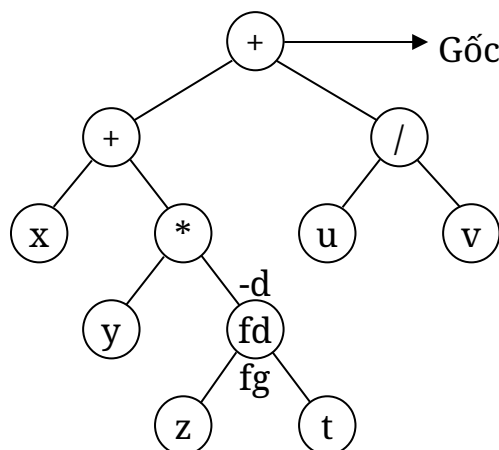
6.1. Định nghĩa và các khái niệm:

6.1.1. Định nghĩa:

- Một nút là một cây. Đó cũng là gốc của cây này.
- Nếu n là một nút và T_1, T_2, \dots, T_k là k cây với các gốc là n_1, n_2, \dots, n_k thì một cây mới sẽ được thành lập bằng cách cho nút n trở thành cha của n_1, n_2, \dots, n_k (được gọi là các cây con của gốc n).



Ví dụ: Cho biểu thức dạng trung tố: $x + y * (z - t) + u / v$. Biểu thức này có thể được biểu diễn dưới dạng cây như sau:



6.1.2. Các khái niệm liên quan:

a. Cấp (bậc - degree):

- Cấp của một nút là số các cây con của nút đó. Suy ra nút có cấp 0 gọi là lá (leaf), ngược lại gọi là nhánh (branch).
- Cấp của một cây là cấp cao nhất của các nút trong cây.

b. Mức (level):

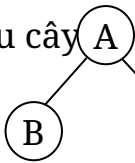
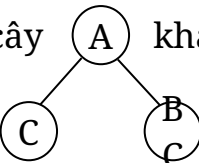
- Gốc có mức 1.
- Một nút có mức i thì nút con của nó có mức $i + 1$.

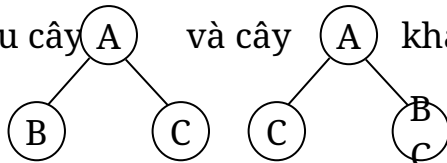
Lưu ý:

- Mức lớn nhất của cây được gọi là chiều cao của cây.
- Nếu có một dãy các nút n_1, n_2, \dots, n_k sao cho n_i là cha của n_{i+1} ($i = \overline{1, k-1}$) thì dãy này được gọi là một đường đi từ n_1 đến n_k , và k được gọi là độ dài đường đi.

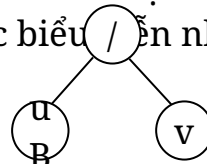
c. Cây có thứ tự:

- Là cây mà thứ tự của các cây con được coi trọng.

Ví dụ: Nếu cây  và cây  khác nhau thì đây là các cây có thứ tự.



Lưu ý: Thường thì thứ tự của các cây con của một nút được tính từ trái sang phải. Ví dụ, biểu thức u / v được biểu diễn như sau:



d. Rừng (forest): Là một tập hợp các cây phân biệt.

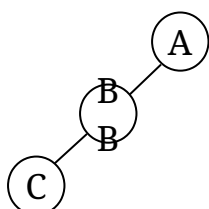
6.2. Cây nhị phân:

6.2.1. Định nghĩa và tính chất:

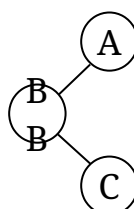
- Cây nhị phân là cây mà tại mỗi nút có tối đa là 2 con.

Nhận xét:

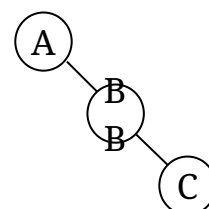
- Cây nhị phân không nhất thiết là cây cấp 2. Ví dụ:
- Một cây cấp 2 thì là cây nhị phân.
- Cây nhị phân là cây có thứ tự.
- Một cây nhị phân được gọi là suy biến nếu nó là cây cấp 1, cụ thể:



Cây lệch trái



Cây zic-zắc



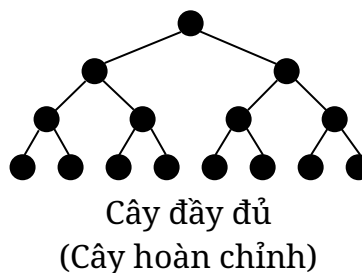
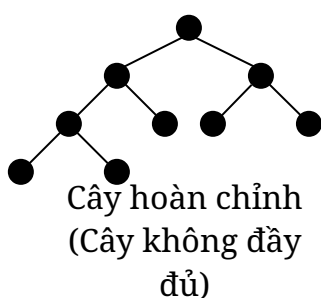
Cây lệch phải

Bổ đề:

- Số lượng tối đa các nút ở mức i trong một cây nhị phân là: 2^{i-1} .
- Số lượng tối đa các nút trên cây nhị phân có chiều cao h là: $2^h - 1$.

Lưu ý:

- Một cây được gọi là *hoàn chỉnh* nếu số nút trên các mức đều đạt tối đa trừ mức cuối cùng.
- Một cây được gọi là *đầy đủ* nếu số nút trên các mức đều đạt tối đa.

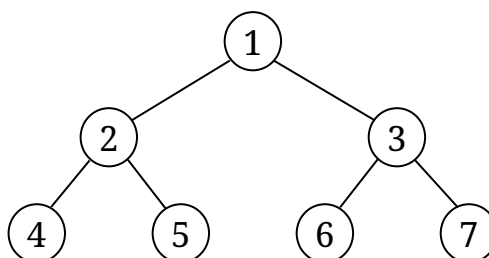


6.2.2. Biểu diễn cây nhị phân:

6.2.2.1. Lưu trữ kế tiếp:

Nếu có một cây nhị phân đầy đủ, ta có thể dễ dàng đánh số các nút trên cây từ mức 1 trở đi theo hướng từ trái sang phải.

Ví dụ:



Nhận xét: Lúc đó các con của nút thứ i có số thứ tự là: $\{2i, 2i + 1\}$

và ngược lại cha của nút thứ j là $\lfloor j/2 \rfloor$ ($j/2$).

Do đó, ta có thể lưu trữ cây với nội dung ở nút thứ i là $V[i]$, bằng cách này ta có thể trực tiếp di chuyển đến mọi nút của cây.

Tuy nhiên đối với một cây không đầy đủ (ví dụ như cây suy biến) thì việc tổ chức lưu trữ theo kiểu này tỏ ra lãng phí (cho một ví dụ).

6.2.2.2. Lưu trữ móc nối:

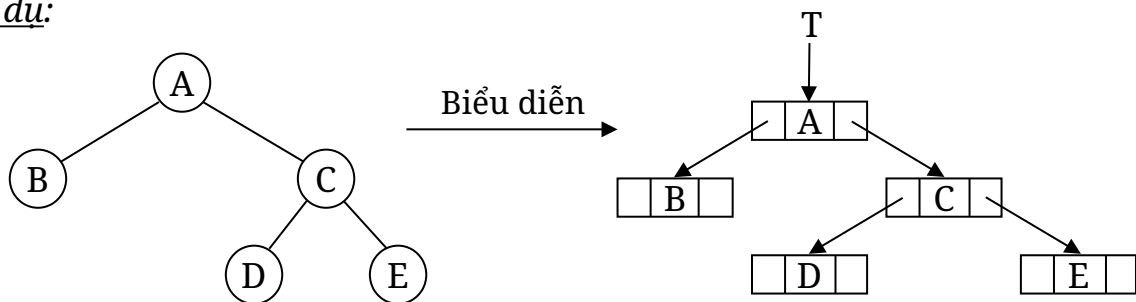
- Ta có thể biểu diễn một nút có 3 phần như sau:

Left	Info	Right
------	------	-------

Trong đó: + Info có thể có nhiều trường.

+ Left, Right: là trường kiểu con trỏ để trỏ tới cây con trái và cây con phải.

Ví dụ:



- Để có thể truy nhập vào các nút trên cây, cần có một con trỏ T trỏ tới nút gốc của cây đó.
- Người ta quy ước nếu cây nhị phân rỗng thì T = NULL.

Nhận xét:

- Tại các nút lá, trường Left và Right có giá trị là NULL.
- Nếu một nút không có cây con bên trái thì trường Left = NULL (tương tự đối với trường Right).

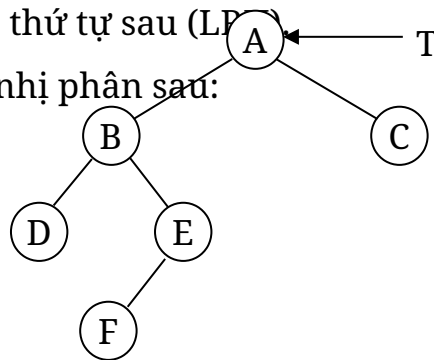
6.2.3. Phép duyệt cây nhị phân:

Phép duyệt cây là phép lần lượt đi qua mọi nút của cây và mỗi nút chỉ đi qua 1 lần (thăm 1 lần). Có 3 phép duyệt cây dựa vào thứ tự duyệt.

Duyệt theo thứ tự trước, thứ tự giữa và thứ tự sau tùy thuộc vào nút gốc (N), cây con trái (L), cây con phải (R), thành phần nào được duyệt trước và thành phần nào được duyệt sau. Chẳng hạn:

- Duyệt theo thứ tự trước có nghĩa là gốc được duyệt trước (NLR).
- Duyệt theo thứ tự giữa (LNR).
- Duyệt theo thứ tự sau (LRN).

Ví dụ 1: Cho cây nhị phân sau:



6.2.3.1. Duyệt theo thứ tự trước:

```
void DuyệtTruoc(T)
    if (T!=NULL )
    {
        cout << T->Info;
        DuyệtTruoc(T->Left);
        DuyệtTruoc(T->Right);
    }
return;
```

➤ Đối với cây ở ví dụ trên, ta có kết quả sau khi duyệt là: ABDEFC.

- Ta có thể khử đệ quy thủ tục này bằng thủ tục sau:

```
void DuyệtTruoc(T)
1. Top=0; Push(S, T);
2. do {
    Pop(S, pt);
    cout << pt->Info;
    if (pt->Right!=NULL) Push(S, pt->Right);
    if (pt->Left!=NULL) Push(S, pt->Left);
} while (top!=0);
return;
```

6.2.3.2. Duyệt theo thứ tự giữa:

```
void DuyệtGiua(T);
    if (T!=NULL)
    {
        DuyệtGiua(T->Left);
        cout << T->Info;
        Duyệtgiua(T->Right);
    }
return;
```

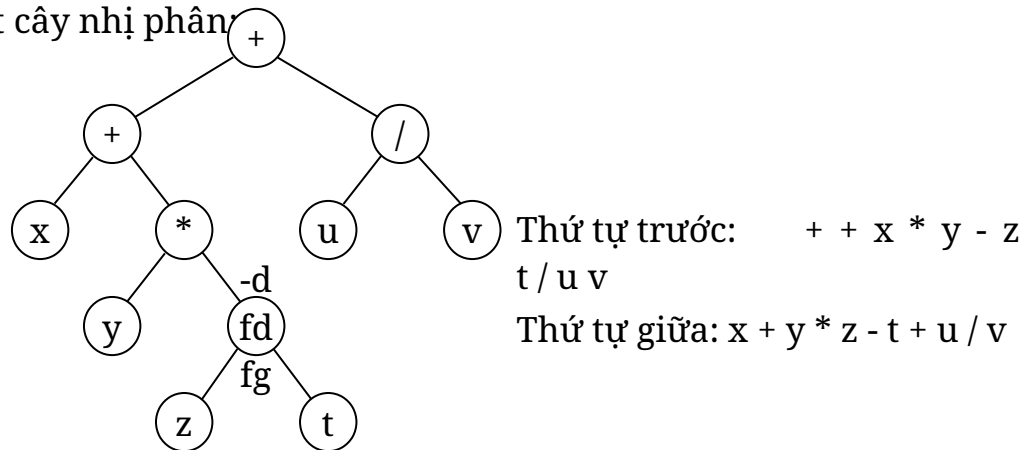
➤ Đối với cây ở ví dụ trên, ta có kết quả sau khi duyệt là: DBFEAC.

6.2.3.3. Duyệt theo thứ tự sau:

```
void DuyệtSau(T);
if T!=NULL
{
    Duyệtsau(T->Left);
    Duyệtsau(T->Right);
    cout << T->Info;
}
return;
```

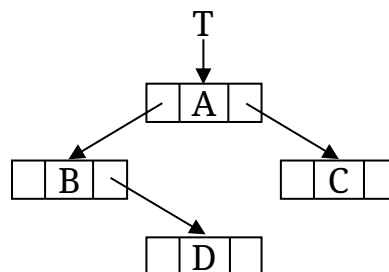
➤ Đối với cây ở ví dụ trên, ta có kết quả sau khi duyệt là: DFEBCA.

Ví dụ 2: Xét cây nhị phân



♣ Bài tập:

1) Tạo cây (sử dụng biểu diễn móc nối) như hình ảnh sau:



Struct Nut

```
{
    Char Info;
    Nut *Left, *Right;
};
Nut *T;

void TaoNut(Nut *p; char X)
{
    p=new Nut;
    p->Info=X;
    p->Left=p->Right=NULL;
}

void TaoCay;
{
    TaoNut(T, 'A');
    TaoNut(T, 'B'); T->Left=p;
    TaoNut(T, 'C'); T->Right=p;
    TaoNut(T, 'D'); T->Left->Right=p;
}

{
    TaoCay;
    .....
}.
}
```

2) Giả sử có cây nhị phân mà gốc trở bởi T. Nhập một xâu từ bàn phím chỉ gồm một trong các chữ R, L để chỉ đường dẫn đến một nút nào đó từ gốc T (R: rẽ phải, L: rẽ trái). Từ đó in nội dung trường Info của nút này.

```
Gets(Xau);
p=T; i=1; Kt=1;//cho KT=true
while (Kt && (i<=strlen(Xau)))
{
    if (p!=NULL )
        if (Xau[i]=='L') p=p->Left;
        else p=p->Right;
    else Kt=0;
    i=i+1;
}
```

```

    }
    if (Kt==1) cout << p->Info;
        else cout << "Đường dẫn sai";

```

3) Tạo một cây (gốc trở bởi T) với dữ liệu về cây được cho trong file văn bản có cấu trúc như sau:

\ <Ký tự> <Các đường dẫn> <ký

Ví dụ:

\ A L B R C

4) Viết thủ tục khử đệ quy thủ tục duyệt giữa và duyệt sau.

5) Nhập vào 2 xâu St1, St2. Trong đó St1 thể hiện phép duyệt trước, St2 thể hiện phép duyệt giữa. Từ đó tạo ra một cây trở bởi T (gợi ý: làm bằng đệ quy).

```

Nut *TaoCay(char *St1,char *St2)
Nut *p;  int p0;
{
    if ((St1!="") && (St2!=""))
    {
        p0=Pos(St1[1], St2);
        p=new Nut;
        p->Info=St1[1];
        p->Left=TaoCay(Copy(St1, 2, p0-1),
                      Copy(St2, 1, p0-1);
        p->Right=TaoCay(Copy(St1, p0+1, strlen(St1)-p0),
                      Copy(St2, p0+1, strlen(St2)-p0));
        TaoCay=p;
    }
    else TaoCay=NULL;
}

{
    Gets(St1); gets(St2);
    T=TaoCay(St1, St2);
    .....
}.

```

6) Chứng minh sự duy nhất của cây khi đã biết kết quả duyệt theo thứ tự trước và thứ tự giữa.

7) Viết lại thủ tục tạo cây trên nhưng có kiểm tra sự mâu thuẫn của xâu St1 và St2.

8) Có xác định được một cây hay không, nếu biết thứ tự duyệt trước (St1) và thứ tự duyệt sau (St2)?

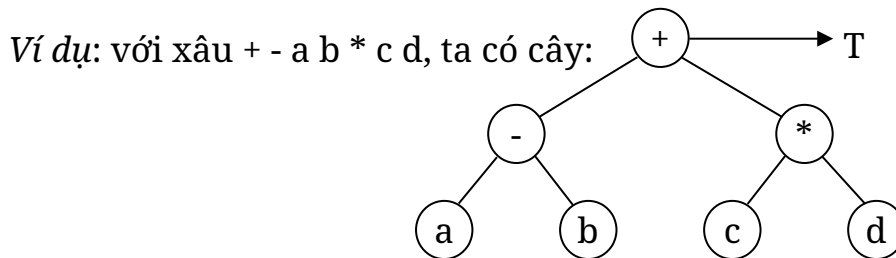
9) Giả sử một phần tử có khai báo:

```
Struct Nut;  
{  
    int      Info: Word;  
    Nut      *Left, *Right;  
};
```

Viết một chương trình tạo cây, biết rằng thứ tự các nút được duyệt theo thứ tự trước lưu trong một mảng int V[0..99]. Và thứ tự duyệt giữa được nhập từ bàn phím dựa vào vị trí trong thứ tự duyệt trước.

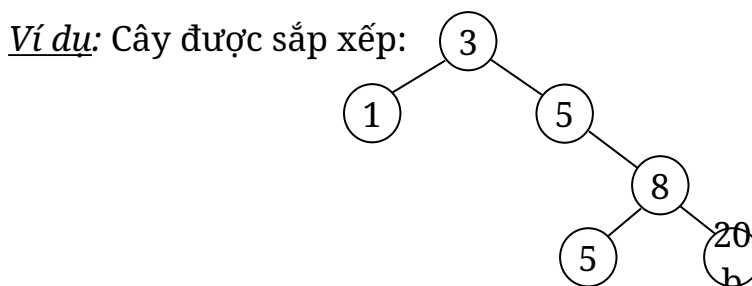
10) Viết chương trình nhập vào một xâu (chỉ chấp nhận các ký tự +, -, *, /, a, b, c, ..., z) biểu diễn biểu thức dạng tiền tố. Từ đó viết chương trình tạo một cây sao cho:

- Gốc trở bởi T.
- Nội dung các nút trong cây là các token của biểu thức trên.
- Phép duyệt cây này theo thứ tự trước chính là nội dung của xâu trên.



❖ Áp dụng cây trong thuật toán sắp xếp:

- *Định nghĩa:* Một cây nhị phân mà mỗi nút của nó chứa một số được gọi là cây được sắp xếp nếu mọi nút trên cây con trái có giá trị bé hơn đỉnh (gốc) và mọi nút trên cây con phải đều lớn hơn hoặc bằng đỉnh (gốc).



- *Bài toán đặt ra:* Viết chương trình nhập vào một số và chèn nó vào 1 cây được sắp xếp. Để giải quyết bài toán này, ta sử dụng thủ tục sau:

```
{ Chèn một nút có giá trị X vào cây được sắp có nút gốc trỏ bởi T. }  
void Chen(T, X)  
{  
    if (T==NULL)  
    {  
        T=new Nut; T->Info=X;  
        T->Left=T->Right=NULL;  
    }  
    else if (X < T->Info) Chen(T->Left, X);  
    else Chen(T->Right, X);  
}
```

Nhận xét: Cây được sắp khi được duyệt theo thứ tự giữa sẽ cho kết quả là dãy số tăng dần.

=> Ta có chương trình chính như sau:

```
void main()  
1. T=NULL;  
2. do  
    {  
        cin >> X;  
        Chen(T, X);  
    }  
    while <còn nhập>;  
3. DuyệtGiua(T);  
}
```

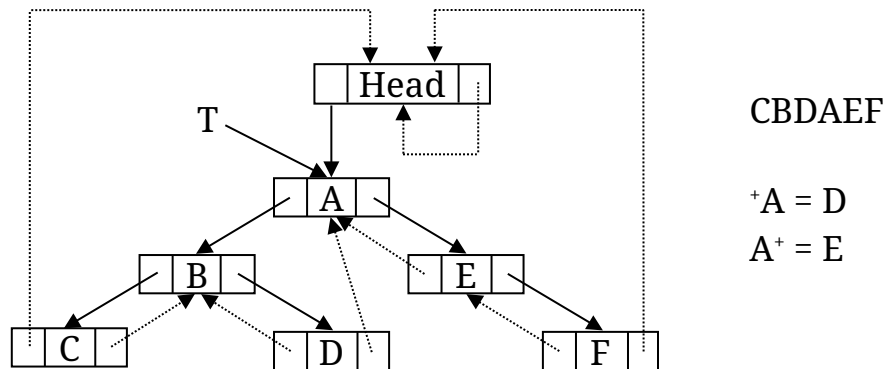
6.2.4. Cây nhị phân nối vòng:

Ta thấy số các trường móc nối (Left và Right) của một cây có giá trị NULL khá nhiều (nếu có n nút thì sẽ có (n+1) con trỏ NULL). Để tận dụng các trường móc nối này, người ta tìm cách cho các con trỏ này trỏ đến các nút quy định mà theo một nghĩa nào đó là để tạo điều kiện cho phép duyệt cây được thuận tiện. Loại móc nối này gọi là nối vòng và cây nhị phân như vậy được gọi là cây nhị phân nối vòng.

Quy ước: Để xây dựng cây nối vòng từ một cây đã cho ta có một số quy ước sau:

- Phép duyệt cây theo thứ tự giữa.
- Cho một nút P và cho một phép duyệt (theo thứ tự giữa). Lúc đó, ta có ký hiệu:
 - ^+P : là nút đứng trước P.
 - P^+ : là nút đứng sau P.

Ví dụ: Cho cây có thứ tự giữa Head CBDAEF Head:



- Với một nút P bất kỳ, nếu:
 - + $p \rightarrow \text{Left} = \text{NULL}$ thì điều chỉnh để $p \rightarrow \text{Left} = ^+P$.
 - + $p \rightarrow \text{Right} = \text{NULL}$ thì điều chỉnh để $p \rightarrow \text{Right} = P^+$.

Nhận xét:

- Bấy giờ rõ ràng máy không thể phân biệt móc nối nào là thực và móc nối nào là giả. Vì vậy, trong quy cách của một nút ta phải thêm vào 2 trường kiểm tra: LType và RType như sau:
 - + Khi $p \rightarrow \text{LType} = 1$ (tương ứng True) thì $p \rightarrow \text{Left}$ là trở thực. Khi $p \rightarrow \text{LType} = 0$ (tương ứng False) thì $p \rightarrow \text{Left}$ là trở giả.
 - + Tương tự đối với RType.
- Ta cũng nhận xét rằng, trong việc thiết lập như ví dụ trên, trường Left nút cực trái (nút C) và trường Right nút cực phải (nút F) vẫn còn NULL. Để tiện dụng người ta đưa vào một nút đầu cây là Head sao cho T là cây con trái của Head, nghĩa là: $\text{Head} \rightarrow \text{Left} = T$ và trường Right của Head trở lại Head nghĩa là: $\text{Head} \rightarrow \text{Right} = \text{Head}$. Và quy định trường Left nút cực trái và trường Right nút cực phải trở tới Head.

➤ Hàm xác định p^+ của p:

```

Nut *Succ(p) //p: Nut
1. q=p->Right;
   if (p->Rtype==0) //p->Rtype =false
   {
       return q; return;
   }
2. while (q->Ltype ==1) q=q->Left; //p->Ltype=true
3. return q;

```

➤ *Hàm xác định ⁺p của p:*

```

Nut *Pred(p) //p: Nut
1. q=p->Left;
   if (p->Ltype ==0)
   {
       return q; return;
   }
2. while (q->Rtype==1) q=q->Right;
3. return q;

```

➤ *Thủ tục duyệt cây nối vòng:*

```

void Duyet cay(Head)
1. p=Head;
2. while (1)
   {
       p=Succ(p);
       if (p==Head) return;
       else cout << p->Info;
   }
return;

```

➤ *Thủ tục thực hiện phép bổ sung một nút vào cây thành cây con trái của nút p cho trên cây nối vòng:*

```

void BoSungTrai(p, X) //Nut moi co noi dung la X
1. Q=new Nut;
   q->Info=X;
   q->Left=p->Left; q->LType=p->LType;
   p->Left=q; p->LType=1;
   q->Right=p; q->RType=0;
2. if (q->LType==1) //p đang tro toi mot canh
   {

```

```

        w=Pred(q);
        w->Right=q;
        w->RType=0;
    }
    return;

```

♣ Bài tập:

Viết chương trình tạo một cây nhị phân nối vòng từ một cây nhị phân đã cho (tham khảo trang 131 cuốn cấu trúc dữ liệu của Nguyễn Trung Trực).

6.3. Cây tổng quát:

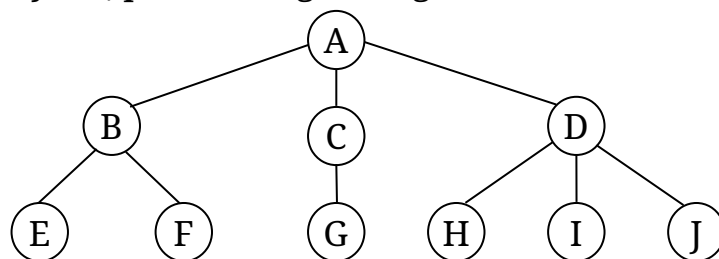
6.3.1. Biểu diễn cây tổng quát:

Đối với cây tổng quát, người ta biểu diễn nó thông qua một cây nhị phân. Cụ thể: Ta để ý rằng, bất kỳ một nút nào đó trên cây tổng quát nếu có thì chỉ có:

- + Một nút con cực trái (con đầu).
- + Một nút “em” cận phải.

Lúc này, cây nhị phân biểu diễn cây tổng quát theo hai quan hệ này được gọi là cây nhị phân tương đương.

Ví dụ:



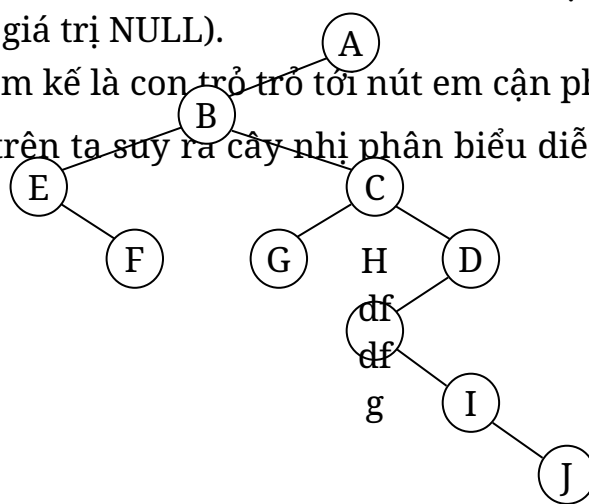
Khi chuyển qua cây nhị phân, một nút có dạng:

Con cả	Info	Em kế
--------	------	-------

Trong đó:

- + Trường con cả là con trỏ trỏ tới nút con cực trái (nếu không có thì nó có giá trị NULL).
- + Trường em kế là con trỏ trỏ tới nút em cận phải.

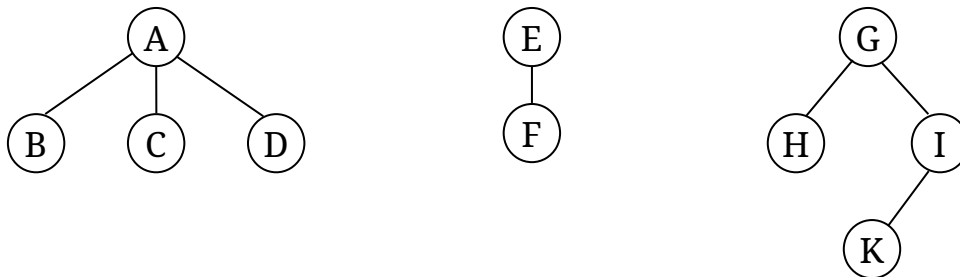
➤ Từ ví dụ trên ta suy ra cây nhị phân biểu diễn cây tổng quát trên là:



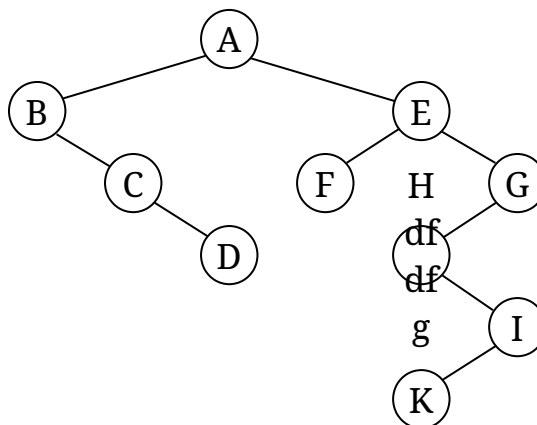
Nhận xét:

- Do nút gốc của cây tổng quát không có nút em nên nút gốc cây nhị phân tương ứng không có cây con phải (trường em kế của nút gốc có giá trị NULL).
- Tuy nhiên, nếu có một rừng cây tổng quát được đánh số thứ tự thì có thể chuyển thành một cây nhị phân (với lưu ý, gốc của cây này có thể xem là em của gốc cây tổng quát khác).

Ví dụ: Cho một rừng có 3 cây tổng quát:



Từ đây ta có thể biểu diễn chúng thành một cây nhị phân như sau:



6.3.2. Phép duyệt cây tổng quát:

Tương tự như cây nhị phân, người ta cũng có phép duyệt cây tổng quát theo:

➤ Thứ tự trước:

- Duyệt gốc.
- Lần lượt duyệt các cây con theo thứ tự trước.

Ví dụ: Xét ví dụ 1, ta có kết quả sau khi duyệt: ABEFCGDHIJ.

Nhận xét: Phép duyệt theo thứ tự trước trong cây tổng quát tương đương với phép duyệt theo thứ tự trước trên cây nhị phân tương ứng (thứ tự trước trên cây nhị phân: ABEFCGDHIJ).

➤ Thứ tự sau:

- Lần lượt duyệt các cây con theo thứ tự sau.
- Duyệt gốc.

Ví dụ: Xét ví dụ 1, ta có kết quả sau khi duyệt: EFBGCHIJDA.

Nhận xét: Phép duyệt theo thứ tự sau trong cây tổng quát tương đương với phép duyệt theo thứ tự giữa trên cây nhị phân tương ứng (thứ tự giữa trên cây nhị phân: EFBGCHIJDA).

Lưu ý: Phép duyệt cây tổng quát thường chỉ xét thứ tự trước và thứ tự sau.

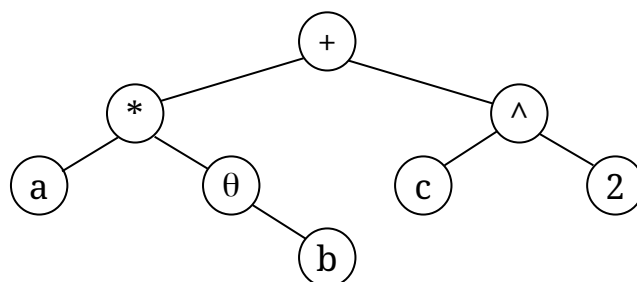
♣ Bài tập:

Cho trước một cây nhị phân biểu diễn một rừng cây tổng quát. Cho biết rừng này có bao nhiêu cây.

6.4. Ứng dụng (Biểu diễn cây biểu thức số học):

Như đã biết, một biểu thức số học với các phép toán 2 ngôi: +, -, *, /, ^ (lũy thừa) được biểu diễn một cách tự nhiên bởi một cây nhị phân. Ta có thể đưa thêm vào phép toán 1 ngôi: θ (phép đổi dấu).

Ví dụ: Cho biểu thức sau: $a * (-b) + c^2$. Ta có cây nhị phân biểu diễn nó như sau:



Với loại cây này, tất cả các toán hạng đều là lá, còn các toán tử thì nằm ở nhánh và có thể biểu diễn bằng một nút như sau:

Left	Type	Right
------	------	-------

Ở đây, trường Info được thay bằng trường Type:

θ).
$$\text{Type} = \begin{cases} 1, 2, 3, 4, 5, 6 \text{ tương ứng với 6 phép toán } (+ - * / ^) \\ 0 \text{ nếu đó là lá.} \end{cases}$$

Như vậy nếu nút lá thì trường Type có giá trị 0 để chỉ biến hoặc hằng tương ứng ở nút đó. Trong trường hợp này, ta lại cho trường Right trở tới địa chỉ trong bảng ký hiệu của biến hoặc hằng đó.

Lưu ý:

Với quy cách như trên thì nút trên cây sẽ lưu trữ loại (Type) phép toán chứ không lưu trữ dấu phép toán. Còn bảng ký hiệu thì được tổ chức để chứa tên của biến (dùng trường Symbol) hay hằng và giá trị của chúng (là trường Value).

➤ Từ đây, ta có thuật toán để tính biểu thức số học được biểu diễn trên một cây nhị phân, có gốc trở bởi E:

```
Struct NutGT{
    char *Symbol;
    float Value;
};
NutGT *F;
NutGT V[100];

float Tinh(E)
switch (E->Type){
    case 0: {
        F=E->Right; return F->Value;
    }
    case 1: return Tinh(E->Left)+Tinh(E->Right);
    case 2: return Tinh(E->Left)-Tinh(E->Right);
    case 3: return Tinh(E->Left)*Tinh(E->Right);
    case 4: return Tinh(E->Left)/Tinh(E->Right);
    case 5: return pow(Tinh(E->Left),Tinh(E->Right));
    case 6: return -Tinh(E->Right);
}
```

return;

♣ Bài tập (Bài thực hành số 3): (Chọn 1 trong 2 đề)

Đề 1:

Ứng dụng cây trong việc tính biểu thức số học, cần giải quyết 3 ý sau:

- Chuyển một biểu thức trung tố thành tiền tố.
- Nhập một biểu thức dạng tiền tố (ví dụ: + - a b * c d).
- Từ đó viết chương trình tạo cây:
 - + Gốc trở bởi T.
 - + Nội dung các nút trong cây là các token của biểu thức trên.
 - + Phép duyệt cây theo thứ tự trước chính là nội dung của biểu thức trên.
 - + Chuyển thuật toán tính giá trị của một biểu thức số học (trong lý thuyết) thành chương trình (hàm Tính) => Nhập 1 xâu dạng trung tố và tính ra kết quả.

➤ Chương trình chính:

```
{
    gets(Xau);
    T=NULL;
    i=0; //i: So Token doc duoc
    Taocay(T);
}
```

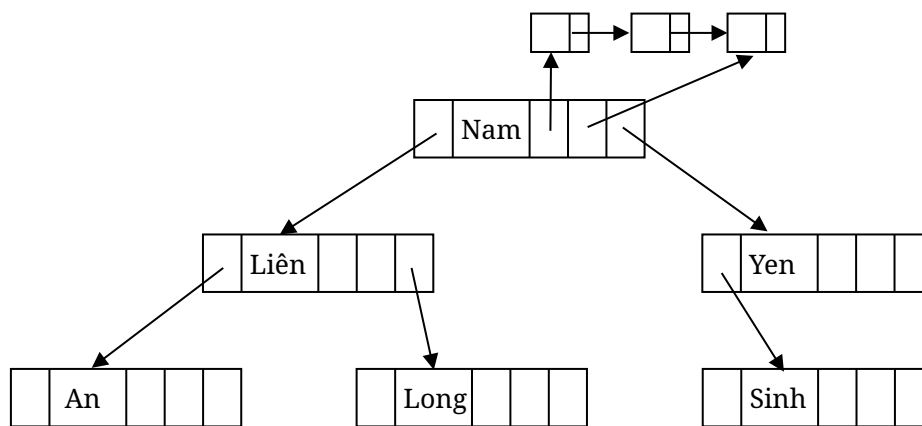
➤ Thủ tục tạo cây:

```
void TaoCay(T)
1. <Đọc một token X (tiếp theo) từ biểu thức tiền tố
   (đọc từ trái sang phải)>;
2. T=new Nut; T->Info=X;
3. if <X là toán hạng>
    T->Left=T->Right=NULL
    else // X = +, -, *, /
    {
        Taocay(T->Left);
        Taocay(T->Right);
    }
return;
```

Đề 2:

Người ta biểu diễn thông tin của một thư viện dưới dạng một cây tìm kiếm nhị phân với khoá tìm kiếm *TenTG* (tên tác giả). Mỗi nút của cây là một bản ghi gồm trường *TenTG* và 4 trường con trỏ:

- Hai con trỏ *Trai* và *Phai* lần lượt trỏ tới các nút con trái và nút con phải.
- Hai con trỏ *Dau* và *Cuoi* lần lượt trỏ tới phần tử đầu và phần tử cuối của một danh sách tuyến tính móc nối dùng để ghi nhận các sách có trong thư viện của tác giả đó. Mỗi phần tử của danh sách này là một bản ghi gồm 2 trường: *TenSach*, *TiepTheo*. Có thể hình dung cây này như hình vẽ sau:



➤ Nút gốc của cây trỏ bởi biến con trỏ *Goc*, ta có khai báo:

```
typedef char St25[25];
struct Sach{
    St25 TenSach;
    Sach *TiepTheo;
};
struct TG {
    TG *Trai,*Phai;
    Sach *Dau,*Cuoi;
    St25 TenTG;
};
TG * Goc;
```

a) Viết hàm: `TG * Nut(TG Goc; St25 Ten);`

Cho kết quả là một con trỏ:

- Bằng *NULL*: khi gốc bằng *NULL*,

- Nếu không thì, trở tới nút có $TenTG = Ten$ nếu nút đó tồn tại,
- Nếu không thì, trở tới một nút trong đó $Ten < TenTG$ và $Trai=NULL$ hoặc
trở tới một nút trong đó $Ten > TenTG$ và $Phai=NULL$.

b) Viết hàm: `TG * NutMoi(St25 Ten;char *TuaDe)`

Cho ta địa chỉ (con trỏ kiểu *TroTG*) của một nút mới thành lập, nhưng chưa được gắn vào cây. Trong đó $TenTG = Ten$ và trong phần tử duy nhất của danh sách tương ứng $TenSach = TuaDe$.

c) Viết thủ tục: `void Bosung(TG *Goc;char Ten[25];
char *TuaDe)`

Cho phép bổ sung tên một tác giả (có tên là *Ten*) với một cuốn sách (có tựa đề là *TuaDe*) vào thư viện trở bởi gốc theo cách sau:

- Nếu tên và tựa đề đều đã có trong thư viện thì không phải làm gì nữa.
- Nếu tên đã có và tựa đề chưa có thì bổ sung tựa đề đó vào cuối danh sách tương ứng với nút có $TenTG = Ten$.
- Nếu tên và tựa đề đều chưa có thì bổ sung một nút mới vào thư viện với $TenTG = Ten$ và $TenSach = Tuade$.

Yêu cầu: Trong chương trình phải làm được các việc:

1. Nhập thông tin cho cây từ một file text có mẫu tương ứng với cây trong hình vẽ trên là như sau:

***Nam**

Pascal

C++

Excel

***Lien**

...

***Yen**

...

***An**

...

***Long**

...

***Sinh**

...

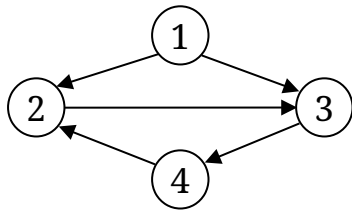
2. Liệt kê tên các tác giả theo thứ tự alphabet.
3. Nhập tên một tác giả, từ đó liệt kê các cuốn sách của tác giả đó.
4. Nhập vào tên một cuốn sách, từ đó cho biết các tác giả đã viết cuốn này.

CHƯƠNG 7: ĐỒ THỊ (GRAPH)

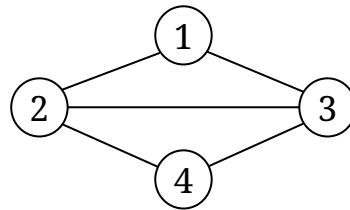
7.1. Định nghĩa và các khái niệm về đồ thị:

- Một đồ thị G gồm một cặp (V, E) trong đó: V là tập các nút (hữu hạn), E là tập các cung (hữu hạn), ký hiệu: $G(V, E)$.

- Người ta thường ký hiệu một cung bởi một cặp đỉnh (V_1, V_2) . Nếu cung $(V_1, V_2) \neq$ cung (V_2, V_1) thì ta có đồ thị định hướng. Ngược lại, nếu thứ tự các nút trên cung không được coi trọng nghĩa là $(V_1, V_2) \equiv (V_2, V_1)$ thì ta có đồ thị không định hướng.



Hình 1: Đồ thị định hướng



Hình 2: Đồ thị không định hướng

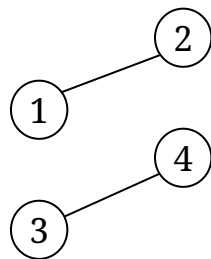
- Cây là một trường hợp đặc biệt của đồ thị.
- Đỉnh V_1 được gọi là lân cận với V_2 nếu tồn tại cung (V_1, V_2) trong đồ thị G .
- Một đường đi từ đỉnh V_p đến đỉnh V_q trong đồ thị G nếu tồn tại các cung: $(V_p, V_{i_1}), (V_{i_1}, V_{i_2}), \dots, (V_{i_n}, V_q)$ thuộc tập E . Số lượng các cung trên đường đi đó được gọi là độ dài của đường đi.

Ví dụ: $(1, 2, 3)$ là đường đi có độ dài 2 (Hình 1).

$(1, 2, 3)$ là đường đi (Hình 2).

- Đường đi đơn là đường đi mà mọi đỉnh trên đó (trừ đỉnh đầu và đỉnh cuối) đều là khác nhau.
- Chu trình: Là một đường đi đơn mà đỉnh đầu và đỉnh cuối trùng nhau.
- Liên thông: Hai đỉnh V_i và V_j được gọi là liên thông nếu tồn tại một đường đi từ V_i tới V_j . Đồ thị G được gọi là liên thông nếu mọi cặp đỉnh phân biệt trong đồ thị đều liên thông.

Ví dụ:



Đồ thị không liên thông

- Một số đồ thị ở mỗi cung người ta gắn thêm một giá trị thể hiện một thông tin nào đó có liên quan tới cung (được gọi là trọng số của cung). Trong trường hợp này, đồ thị được gọi là đồ thị có trọng số.

7.2. Biểu diễn đồ thị:

7.2.1. Biểu diễn bằng ma trận lân cận (ma trận kề):

Xét một đồ thị $G(V, E)$ với V gồm có n đỉnh ($n \geq 1$) mà giả sử các đỉnh đã được đánh số thứ tự theo một quy định nào đó. Ma trận lân cận A biểu diễn đồ thị G là một ma trận vuông có kích thước $n \times n$.

Ví dụ 1: $A = \begin{bmatrix} \text{đỉnh 1} & \text{đỉnh 2} \\ \text{đỉnh 2} & \text{đỉnh 1} \\ \text{đỉnh 3} & \text{đỉnh 4} \\ \text{đỉnh 4} & \text{đỉnh 3} \\ \text{đỉnh 5} & \text{đỉnh 5} \end{bmatrix}$

Nhận xét:

- Các phần tử của ma trận chỉ có giá 0 hoặc 1. Nếu tồn tại một cung (i, j) thì $a_{ij} = 1$, ngược lại thì $a_{ij} = 0$.
- Đối với đồ thị không định hướng thì ma trận A là đối xứng. Còn đối với đồ thị định hướng thì ma trận A không đối xứng.

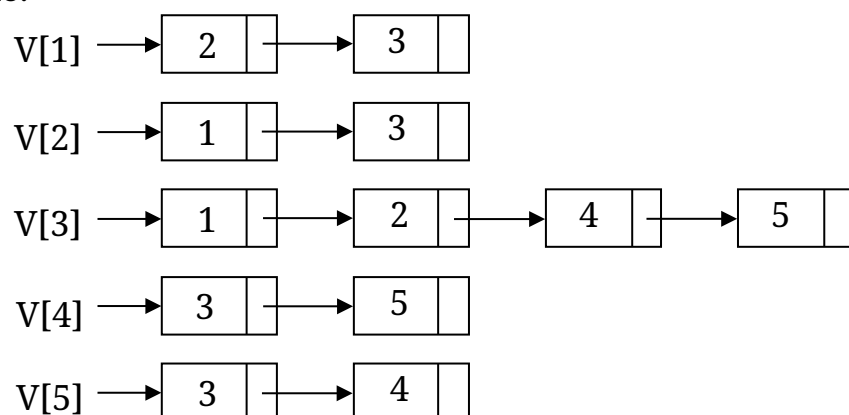
Ví dụ 2: $A = \begin{bmatrix} \text{đỉnh 1} & \text{đỉnh 2} \\ \text{đỉnh 2} & \text{đỉnh 1} \\ \text{đỉnh 3} & \text{đỉnh 4} \\ \text{đỉnh 4} & \text{đỉnh 3} \\ \text{đỉnh 5} & \text{đỉnh 5} \end{bmatrix}$

Lưu ý: Đối với đồ thị có trọng số có thể biểu diễn ma trận này bằng cách thay số 1 bởi trọng số của cung tương ứng.

7.2.2. Biểu diễn bằng danh sách lân cận (danh sách kề):

Trong cách biểu diễn này, n hàng của ma trận lân cận được thay đổi bởi n danh sách móc nối.

Ví dụ: Xét đồ thị không định hướng trong ví dụ 1 ở trên, ta có các danh sách kề:



Nhận xét:

- Mỗi đỉnh của G có một danh sách tương ứng. Các nút ở trong danh sách i (trở bởi V[i]) biểu diễn các đỉnh lân cận của nút i. Mỗi nút có dạng:

Đỉnh	Tiếp
------	------

- Mỗi danh sách i có một nút đầu danh sách (V[i]), các nút này thường được tổ chức thành một mảng để có thể truy cập được nhanh.

Lưu ý: Các nút trong từng danh sách thông thường sắp xếp theo thứ tự.

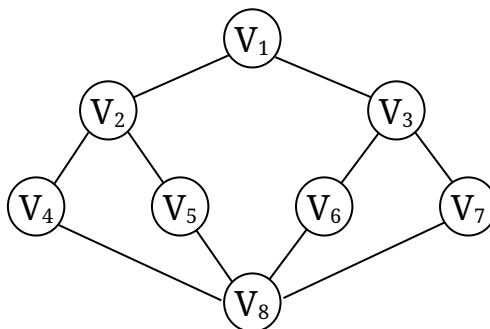
7.3. Phép duyệt một đồ thị:

7.3.1. Tìm kiếm theo chiều sâu:

Xét đồ thị không định hướng và liên thông, phép tìm kiếm theo chiều sâu được thực hiện như sau:

Đầu tiên thăm đỉnh V, sau đó thăm đỉnh W (đỉnh này chưa được thăm) là lân cận của V. Bấy giờ từ W, một phép tìm kiếm theo chiều sâu xuất phát từ W lại được thực hiện. Khi một đỉnh U vừa (đã) được thăm mà mọi đỉnh lân cận của nó được thăm rồi thì ta sẽ quay ngược lên đỉnh gần đây vừa được thăm (đây là thuật toán quay lui). Phép tìm kiếm sẽ kết thúc khi không còn một nút nào chưa được thăm mà vẫn có thể tới được một nút đã được thăm.

Ví dụ:



Thứ tự duyệt như sau:

V1, V2, V4, V8, V5, V7, V3, V6

➤ Từ đó ta có thể xây dựng thuật toán của phép duyệt này là như sau:

```
void Tim_Kiem_Sau(V)
```

```
1. Visited[V]=1;
```

```
   cout << V;
```

```
2. For <Mỗi đỉnh W là lân cận của V>
```

```
   if Visited[W]==0
```

```
       Tim_Kiem_Sau(W);
```

return;

➤ *Chương trình chính:*

```
For (i=1;i<=n;i++)    // n là số nút tối đa của mảng
    Visited[i]=0;
Tim_Kiem_Sau(1);    // Giả sử đồ thị có đỉnh 1
=> Kết quả in ra sẽ là những đỉnh liên thông với đỉnh 1.
```

Lưu ý:

- Trong thủ tục tìm kiếm sâu, ở bước 1 ta có thể bổ sung thêm lệnh chứng tỏ nút V được thăm (ví dụ, lệnh cout << V). Lúc này ở chương trình chính chỉ cần thực hiện các lệnh:
 - Khởi tạo các phần tử của mảng Visited bằng 0.
 - Gọi thủ tục Tim_Kiem_Sau(1).
- Chương trình này chỉ duyệt qua tất cả các nút liên thông với nút 1 mà thôi.
- Phép duyệt cây theo thủ tục tìm kiếm theo chiều sâu tức là duyệt theo thứ tự trước (đối với cây).

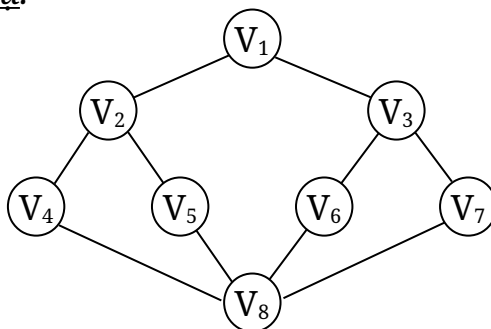
♣ **Bài tập:**

Viết chương trình bằng C++ thể hiện việc tìm kiếm sâu trong một đồ thị bằng 2 cách biểu diễn đồ thị (ma trận kề, danh sách kề).

7.3.2. Tìm kiếm theo chiều rộng:

Phép tìm kiếm theo chiều rộng cũng xuất phát từ một đỉnh V nào đó nhưng khác với phép tìm kiếm theo chiều sâu ở chỗ: các đỉnh là lân cận của V mà chưa được thăm sẽ được thăm kế tiếp nhau rồi mới đến các đỉnh chưa được thăm là lân cận lần lượt của các đỉnh này...

Ví dụ:



Thứ tự duyệt như sau:

V1, V2, V3, V4, V5, V6, V7, V8

➤ Thuật toán tìm kiếm theo chiều rộng sẽ dựa vào nguyên tắc hàng đợi (Queue):

```
void Tim_Kiem_Rong(V)
1. Visited[V]=1;
2. <Khởi tạo hàng đợi Q rỗng>
   Insert_Queue(Q, V);    // cout << V;
3. do
   {
       Delete_Queue(Q, V);
       For <Mỗi W lân cận của V>
           if (Visited[W]==0)
           {
               Insert_Queue(Q, W);
               Visited[W]=1; // cout << W;
           }
   }
while <Queue chưa rỗng>;
return;
```

Nhận xét:

- Tìm kiếm theo chiều rộng sử dụng cấu trúc hàng đợi (Queue). Còn tìm kiếm theo chiều sâu sử dụng Stack (đệ quy).
- Cả 2 thuật toán này đều có độ phức tạp tính toán $O(n^2)$.

♣ Bài tập:

Viết chương trình bằng C++ thể hiện việc tìm kiếm theo chiều rộng bằng 2 cách biểu diễn đồ thị.

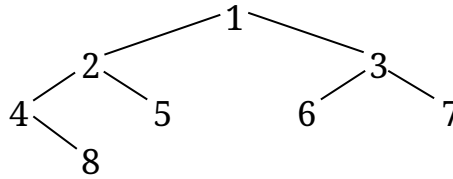
7.4. Cây khung và cây khung với giá cực tiểu:

Định nghĩa: Khi một đồ thị $G(V, E)$ liên thông thì một phép tìm kiếm theo chiều sâu hay chiều rộng xuất phát từ một đỉnh nào đó sẽ cho phép thăm được mọi đỉnh của đồ thị. Trong trường hợp này, các cung của E sẽ được phân thành 2 tập:

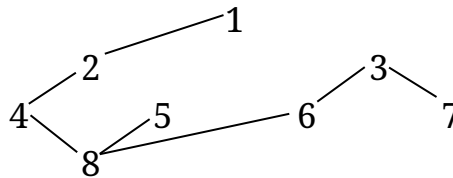
- Tập T bao gồm tất cả các cung được dùng tới hoặc được duyệt qua trong phép tìm kiếm.
- Tập B bao gồm các cung còn lại.

Lúc này, tất cả các cung trong tập T cùng với các đỉnh tương ứng tạo thành một cây khung.

Ví dụ: Cây khung T ứng với ví dụ trong tìm kiếm theo chiều rộng như sau:



Cây khung T ứng với ví dụ trong tìm kiếm theo chiều sâu như sau:



Nhận xét:

- Một cây khung T cho ta một đồ thị liên thông nhưng không tồn tại chu trình nào bên trong đồ thị này.
- Khi ta bổ sung thêm một cung bất kỳ vào một cây khung thì sẽ xuất hiện một chu trình.
- Đồ thị có n đỉnh thì cây khung có $n-1$ cung.

Ứng dụng (Xác định cây khung với giá cực tiểu):

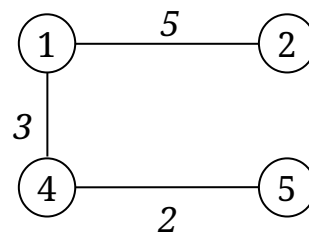
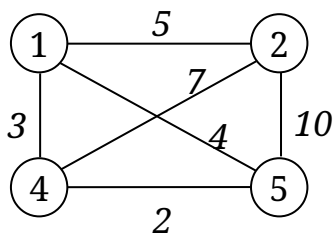
Xét bài toán sau:

Giả sử cho 1 đồ thị liên thông có trọng số, vấn đề được đặt ra: Xác định cây khung với giá cực tiểu (cây khung mà tổng các trọng số trên cây khung đó là nhỏ nhất so với các cây khung khác).

Giải quyết: Ta có thể sử dụng thuật toán của Kruscal.

➤ Ý tưởng:

- Các cung được xét để đưa vào T dựa vào thứ tự không giảm của trọng số tương ứng của chúng. Một cung được đưa vào T nếu nó không tạo nên chu trình với các cung đã có ở trong T.



- Ở đây, nếu đồ thị có n đỉnh thì chỉ cần bổ sung $n-1$ cung.

➤ Thuật toán:

```

void Kruscal(G)
1. T=  $\emptyset$ ; //T là tập chứa các cung
2. while (|T|<n-1)
    {
        <Chọn 1 cung (V, W)  $\in$  E có giá trị bé nhất>;
        <Loại (V, W) khỏi E>;
        if ( <(V, W) không tạo nên chu trình trong T> )
            T = T  $\cup$  {(V, W)} //Bổ sung cung (V, W) vào T
    }
return;

```

CHƯƠNG 8: SẮP XẾP

8.1. Đặt vấn đề:

Sắp xếp là quá trình bố trí lại các phần tử của một tập đối tượng nào đó theo một thứ tự nhất định. Yêu cầu về sắp xếp thường xuyên xuất hiện trong tin học nhằm giúp quản lý dữ liệu được dễ dàng.

Các thuật toán sắp xếp được phân chia thành 2 nhóm chính:

- + Sắp xếp trong: Toàn bộ dữ liệu sắp xếp được đưa vào bộ nhớ trong, do đó dữ liệu không nhiều lắm nhưng ngược lại thời gian sắp xếp nhanh.
- + Sắp xếp ngoài: Một phần dữ liệu cần sắp xếp được đưa vào bộ nhớ trong, phần còn lại được lưu trữ ở bộ nhớ ngoài, do đó có thể sắp xếp dữ liệu với khối lượng lớn nhưng tiến trình sắp xếp sẽ chậm hơn.

Nói chung, dữ liệu có thể xuất hiện dưới nhiều dạng khác nhau. Ở đây ta quy ước tập đối tượng được sắp xếp là tập các bản ghi. Tuy nhiên, khi sắp xếp, thường người ta chỉ quan tâm đến giá trị của 1 trường nào đó (gọi là trường khoá) và việc sắp xếp được tiến hành dựa vào trường khoá này. Để đơn giản, ở đây ta xem 1 bản ghi chỉ chứa 1 trường dữ liệu có kiểu số và thứ tự sắp xếp theo chiều tăng.

8.2. Một số phương pháp sắp xếp đơn giản:

8.2.1. Sắp xếp kiểu lựa chọn:

➤ Nguyên tắc:

Tại mỗi bước i ta chọn trong dãy a_{i+1}, \dots, a_n các phần tử lớn hơn a_i , sau đó thực hiện hoán đổi vị trí của chúng với a_i (sao cho a_i có giá trị nhỏ nhất ($i = 1..n$)).

➤ Thuật toán:

```
void Selection_Sort(a, n)
    For (i=1; i<=n-1; i++)
        For (j=i+1; j<=n; j++)
            if (a[i]>a[j])
                <Đổi chỗ a[i] và a[j]>;
return;
```

8.2.2. Sắp xếp kiểu chèn:

➤ *Nguyên tắc:* Tương tự như khi sắp bài tiến lên. Chia làm 2 trường hợp:

- Kết hợp việc sắp xếp với việc nhập dữ liệu:

```
void SapXep(a, n)
    For (i=1; i<=n; i++)
    {
        cin >> a[i];
        Chen(a[i]);
    }
return;
```

Trong đó, ta có thủ tục *Chen(X)* như sau:

```
void Chen(X)
    if (i>1)
    {
        j=1;
        while ((a[j]<=X) && (j<=i-1)) j=j+1;
        if (j<i)
        {
            For (k=i;k>=j+1;k--)
                a[k]=a[k-1];
            a[j]=X;
        }
    }
return;
```

- Sắp xếp từ mảng đã có dữ liệu:

```
void Insert_Sort(a, n)
Const VoCuc = 106;
1. a[0]= -VoCuc;
2. For (i=1;i<n;i++)
    {
        X=a[i]; j=i-1;
        while (x<a[j])
        {
            a[j+1]=a[j]; j=j-1;
        }
        a[j+1]=X;
    }
```

```
return;
```

Lưu ý: Ý tưởng của thuật toán này có thể thực hiện dễ dàng hơn nếu sử dụng danh sách móc nối để lưu dãy số này.

8.2.3. Sắp xếp kiểu nổi bọt:

```
void Bubble_Sort(a, n)
    For (i=1; i<n; i++)
        For (j=n; j>=i+1; j--)
            if (a[j]<a[j-1])
                <Đổi chỗ a[j] và a[j-1]>;
return;
```

Nhận xét: Cả 3 thuật toán trên đều có độ phức tạp tính toán là $O(n^2)$.

8.3. Sắp xếp kiểu phân đoạn (Sắp xếp nhanh - quick sort):

➤ Nguyên tắc:

Chọn ngẫu nhiên một phần tử X của dãy (chẳng hạn phần tử đầu tiên) và cố gắng phân chia dãy này thành 3 dãy con liên tiếp nhau:

- + Dãy 1: Gồm những phần tử nhỏ hơn X.
- + Dãy 2: Gồm những phần tử bằng X.
- + Dãy 3: Gồm những phần tử lớn hơn X.

Sau đó áp dụng lại thuật toán này cho dãy con thứ nhất và dãy con thứ ba (dãy con này có số phần tử lớn hơn 1).

```
void Quick_Sort(a, Be, Lon)
    if (Be<Lon)
    { i=Be+1; j=Lon; X=a[Be];
      1. do
        { while ((a[i]<X) && (i<=Lon)) i=i+1;
          if (i>Lon)
            { <Đổi chỗ a[Be] và a[Lon]>;
              Quick_Sort(a, Be, Lon-1); return;
            }
          while (a[j]>X) j=j-1;
          if (j=Be )
            { Quick_Sort(a, Be+1, Lon);
              return;
            }
          if (i<=j)
            { <Đổi chỗ a[j] và a[i]>;
              i=i+1;
              j=j-1;
            }
        }
    }
```

```

    }
    while (i<=j);
    <Đổi chỗ a[Be] và a[j]>;
2. if (Be<j-1) Quick_Sort (a, Be, j-1);
   if (Lon>i) Quick_Sort (a, i, Lon);
}
return;

```

Lưu ý: Tại chương trình chính, để sắp xếp mảng a từ phần tử thứ nhất đến phần tử thứ n thì ta gọi thủ tục sắp xếp như sau: Quick_Sort (a, 1, n);

- Người ta chứng minh được trong trường hợp xấu nhất, thuật toán này có độ phức tạp là $O(n\log_2 n)$ (xem sách). Do đó, với n khá lớn thuật toán Quick sort tỏ ra hữu hiệu hơn các thuật toán đơn giản.

8.4. Sắp xếp kiểu vun đống (Heap sort):

- *Nguyên tắc:* Với phương pháp sắp xếp này dãy số được lưu ở trong mảng sẽ được coi như là cấu trúc của cây nhị phân hoàn chỉnh.
- Đầu tiên, cây nhị phân biểu diễn sẽ được sắp xếp để tạo thành một đống (heap). Ta gọi đó là giai đoạn tạo đống (đống là cây nhị phân hoàn chỉnh mà mỗi nút được gán một giá trị sao cho nút cha luôn có giá trị lớn hơn hoặc bằng nút con). Bây giờ giá trị ở gốc sẽ là các khoá lớn nhất (gọi là khoá trội).
- Sau đó, các động tác sau đây sẽ được lặp đi lặp lại nhiều lần cho đến khi cây chỉ còn lại một lá: Đưa khoá trội về vị trí thực của nó (bằng cách đổi chỗ cho khoá ở cuối đống đang xét), sau đó vun lại đống đối với cây gồm các khoá còn lại.

Lưu ý:

Vấn đề được đặt ra là: cần xây dựng một thuật toán để điều chỉnh một cây thành một đống với giả thiết rằng cây con trái và cây con phải của gốc đã là đống.

Một cách tổng quát, ta có thuật toán để điều chỉnh lại cây con tại i, biết rằng cây con trái ($2i$) và cây con phải ($2i+1$) đã là đống (giả sử đống a có n phần tử):

```

void DieuChinh(a, i, n)
1. Key=a[i];
   j=2*i; // j: chỉ số chỉ tới cây con trái của i
2. while (j<=n)

```

```

{
    if ((j<n) && (a[j]<a[j+1]))
        j=j+1;          /*Chọn j ở vị trí con của i nhưng
                           có giá trị lớn nhất */
    if (Key>=a[j])        /*Cần thiết cho lần thực
                           hiện sau của vòng lặp*/
    {
        a[j/2]=Key;
        return;
    }
    a[j/2]=a[j];
    j=2*j;
}
3. a[j/2]=Key;
return;

```

➤ *Lúc này thủ tục sắp xếp theo kiểu vun đống như sau:*

```

void Heap_Sort(a, n);
1. //Giai đoạn 1
   For (i=n/2;i>=1;i--)
       DieuChinh(a, i, n);    /*1 lá có thể xem như là đống
                               cho nên thủ tục này thực hiện
                               từ trên lá trở lên*/
2. //Giai đoạn 2
   For (i=n-1;i>=1;i--)
   {
       <Đổi chỗ a[1] và a[i+1]>;
       Dieuchinh(a, 1, i);
   }
return;

```

Lưu ý: Người ta cũng chứng minh được rằng độ phức tạp của thuật toán này là $O(n\log_2 n)$.

8.5. Sắp xếp kiểu trộn (Merge sort):

- Phép hoà nhập 2 đường. Xét bài toán:

Giả sử ta có mảng X chia làm 2 phần đã được sắp xếp: $(X_b, X_{b+1}, \dots, X_m)$ và $(X_{m+1}, X_{m+2}, \dots, X_n)$. Viết thuật toán tạo ra mảng Z có chỉ số từ b tới n $(Z_b, Z_{b+1}, \dots, Z_n)$ được sắp xếp.

```

void Tron(X, b, m, n, Z);
1. i=b;  j=m+1;  k=b;
2. while ((i<=m) && (j<=n))

```

```

    {
        if (X[i]<=X[j] )
        {
            Z[k]=X[i];  i=i+1;
        }
        else
        {
            Z[k]=X[j];  j=j+1;
        }
        k=k+1;
    }
3. if i>m  (Z[k], ..., Z[n])=(X[j], ..., X[n]);
   else (Z[k], ..., Z[n])=(X[i], ..., X[m]);
return;

```

- Sắp xếp kiểu hòa nhập 2 đường trực tiếp: Sau đây là thủ tục thực hiện một bước sắp xếp kiểu trộn bằng cách trộn từng cặp kế cận nhau có độ dài là L từ mảng X sang mảng Y, với n là số phần tử ở trong X.

```

void Chuyen(X, Y, n, L)
1. i=1;
2. while (i<=n-2*L+1)
    {
        Tron(X, i, i+L-1, i+2*L-1, Y);  // i+2*L-1 <= n
        i=i+2*L;
    }
3. {Trộn phần còn dư với phần trước}
   if (i+L-1>n) (Y[i], ..., Y[n])=(X[i], ..., X[n])
   else Tron(X, i, i+L-1, n, Y);
return;

```

=> Từ đây ta có thể suy ra thủ tục sắp xếp theo kiểu trộn như sau:

```

void Merge_Sort(X, Y, n)
1. L=1;
2. while (L<n)
    {
        Chuyen(X, Y, n, L);
        L=L*2;
        X=Y;
    }
return;

```

Lưu ý:

- Người ta chứng minh được rằng độ phức tạp của thuật toán này là $O(n \log_2 n)$. Tuy nhiên, do phải tạo ra mảng Y nên phương pháp này tốn bộ nhớ trong hơn so với 2 thuật toán trên.
- Thuật toán này thường được áp dụng cho việc sắp xếp ngoài (có kết hợp với file).

CHƯƠNG 9: TÌM KIẾM

9.1. Bài toán tìm kiếm:

Tìm kiếm là một yêu cầu rất thường xuyên trong đời sống hàng ngày cũng như trong tin học. Để đơn giản ta xét bài toán tìm kiếm như sau:

Cho một dãy số gồm các phần tử a_1, a_2, \dots, a_n . Cho biết trong dãy này có phần tử nào có giá trị bằng X (cho trước) hay không?

9.2. Tìm kiếm tuần tự:

Thuật toán tìm kiếm tuần tự có sử dụng một biến logic, biểu thị một phần tử có tồn tại trong dãy cần tìm hay không. Ở đây ta cũng có thể giải quyết theo cách khác:

```
int TimKiemTT(a, n, X)
1. i=1; a[n+1]=X;
2. while (a[i]!=X) i=i+1;
   if (i==(n+1)) return 0
   else return i;
```

=> Hàm này sẽ trả về giá trị là một chỉ số i nào đó trong dãy nếu tìm thấy, ngược lại hàm sẽ trả về giá trị 0.

Lưu ý: Thuật toán này có độ phức tạp là $O(n)$.

9.3. Tìm kiếm nhị phân:

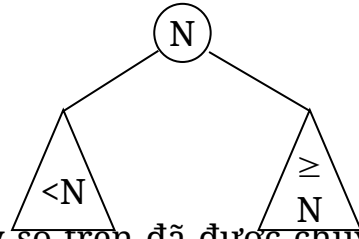
Với giả thiết ban đầu dãy đã được sắp theo thứ tự tăng dần. Thuật toán tìm kiếm nhị phân bằng đệ quy ta đã biết trong phần đệ quy. Tuy nhiên ta có thể khử đệ quy thuật toán này như sau:

```
int TKNP(a, n, X);
1. Be=1; Lon=n;
2. while (Be<=Lon)
{
    Giua=(Be+Lon)/ 2;
    if (a[Giua]==X) return Giua;
    if (a[Giua]<X) Be=Giua+1;
    else Lon=Giua-1;
}
3. return 0;
```

Lưu ý: Thuật toán này có độ phức tạp là $O(\log_2 n)$.

9.4. Cây nhị phân tìm kiếm:

Cây nhị phân tìm kiếm là cây được sắp xếp mà ta đã bàn đến.



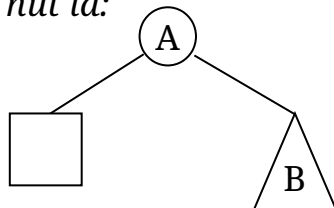
Bài toán: Giả sử dãy số trên đã được chuyển vào cây nhị phân tìm kiếm mà nút gốc được trở bởi T. Vấn đề đặt ra: Viết một hàm $CNPTK(T, X)$ trả về giá trị NULL nếu không có nút nào mà trường *Info* có giá trị bằng X, ngược lại cho kết quả là con trỏ trở vào phần tử đó.

```
Nut * CNPTK(T, X)
1. q=T;
2. while (q!=NULL)
{
    if (q->Info == X) break;
    if (q->Info < X) q=q->Right;
    else q=q->Left;
}
3. return q;
```

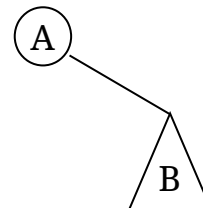
Lưu ý: Khi đã có sẵn cây nhị phân tìm kiếm, thuật toán bổ sung một nút vào cây này thì ta đã xét. Nếu cần loại bỏ một nút trong cây nhị phân tìm kiếm, ta xét các trường hợp sau:

□ : Chỉ nút cần xóa △ : Cây con

1. i) Xoá nút lá:

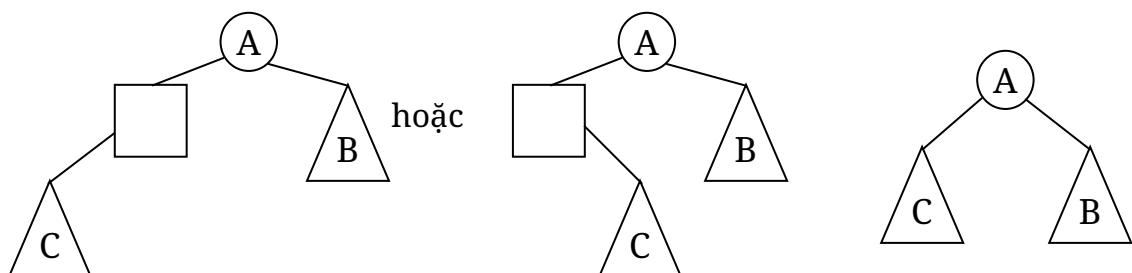


Trước khi xóa



Sau khi xóa

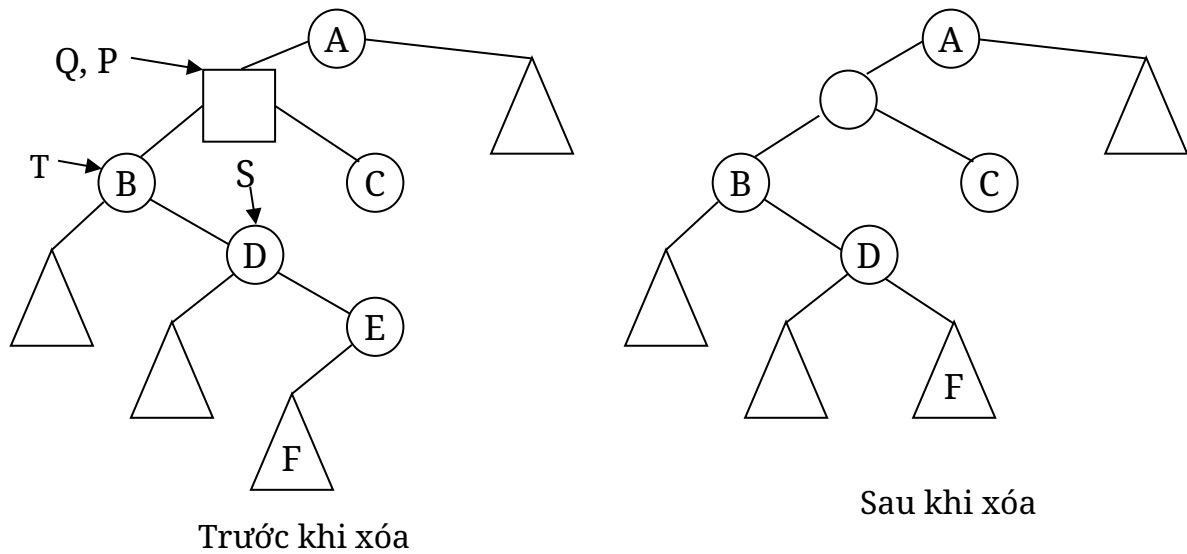
ii) Xoá nút nửa lá:



Sau khi xóa 8

Trước khi xóa

- Xóa 1 nút không là lá, không là nửa lá:



➤ *Thuật toán:* Giả sử ta đã có hàm bố $Bo(p)$:

```
void XoaNut(Q)
1. //Xử lý trong trường hợp nút lá và nửa lá
   p=Q;
   if (p->Left=NULL)
   {
       R=Bo(Q); R->Left=p->Right;
       Free(p); return;
   }
   if (p->Right==NULL)
   {
       R=Bo(Q); R->Left=p->Left;
       Free(p); return;
   }
   //Trường hợp Q là con phải của bố Q thì xử lý tương tự
2. T=p->Left;
   if (T->Right==NULL)
   {
       R=Bo(Q); R->Left=T;
       T->Right=p->Right;
       Free(p); return;
   }
   S=T->Right;
```

```

while (S->Right!=NULL)
{
    T=S;  S=S->Right;
}
S->Right=p->Right;  T->Right=S->Left;
S->Left=p->Left;  R=Bo(Q);  R->Left=S;
Free(p);
return;

```

TÀI LIỆU THAM KHẢO

- [1] Cấu trúc dữ liệu và thuật toán (Đỗ Xuân Lôì)
- [2] Lập trình nâng cao bằng PASCAL với các cấu trúc dữ liệu (Larry Hoff - Lê Minh Trung dịch) - Tập 2
- [3] Cẩm nang thuật toán (Robert Sedgewick) - 2 tập
- [4] The Art of Computer Programming (donald Knuth)
- [5] Algorithm + Data Structure = Program (Niklaus Wirth)
- [6] Cấu trúc dữ liệu và thuật toán, Đinh Mạnh Tường, Nhà Xuất bản ĐHQG Hà Nội, 2008
- [7] T. Cormen, C. E. Leiserson, R. L. Rivest, Introduction to Algorithms, Cambridge, MIT Press, 2nd Edition, 2005