Chương 4 DANH SÁCH TUYẾN TÍNH

Trong chương này, chúng ta sẽ nghiên cứu danh sách tuyến tính, một trong các mô hình dữ liệu quan trọng nhất, được sử dụng thường xuyên trong việc cài đặt các bài toán ứng dụng. Các phương pháp cài đặt danh sách khác nhau sẽ được xem xét. Hai kiểu dữ liệu trừu tượng đặc biệt quan trọng là ngăn xếp (Stack) và hàng đợi (Queue) sẽ được nghiên cứu. Chương này cũng sẽ trình bày một số ứng dụng phổ biến của danh sách.

1. KHÁI NIỆM DANH SÁCH TUYẾN TÍNH

1.1. Khái niệm danh sách

Về mặt toán học, danh sách là một dãy hữu hạn các phần tử thuộc cùng một lớp đối tượng nào đó. Chẳng hạn, danh sách sinh viên của một lớp, danh sách các số nguyên, danh sách các báo xuất bán hàng ngày ở thủ đô, v.v...

Giả sử L là một danh sách có n phần tử (n >= 0).

L = (a1, a2,..., an)

Ta gọi n là độ dài của danh sách. Nếu n >= 1 thì a1 được gọi là phần tử đầu tiên, an được gọi là phần tử cuối cùng của danh sách L. Nếu n = 0 thì danh sách L được gọi là danh sách rỗng.

Một tính chất quan trọng của danh sách là các phần tử của nó được sắp tuyến tính: nếu n > 1 thì phần tử a_i "đi trước" phần tử a_{i+1}. Ta gọi a_i (i = 1, 2,..., n) là phần tử ở vị trí thứ i của danh sách. Nghĩa là, một danh sách mà quan hệ lân cận giữa các phần tử được hiển thị ra thì danh sách đó được gọi là danh sách tuyến tính.

1.2. Các phép toán trên danh sách

Khi mô tả một mô hình dữ liệu, chúng ta cần xác định các phép toán có thể thực hiện trên mô hình toán học được dùng làm cơ sở cho mô hình dữ liệu. Có rất nhiều phép toán trên danh sách. Trong các ứng dụng, thông thường chúng ta chỉ sử dụng một nhóm các phép toán nào đó. Sau đây là một số phép toán cơ bản trên danh sách tuyến tính.

Giả sử L là một danh sách, các phần tử của nó có kiểu ***Item***, k là vị trí của một phần tử trong danh sách. Các phép toán sẽ được mô tả bởi các hàm sau đây:

1. Khởi tạo danh sách rỗng

void Initialize(***List*** *L);

2. Xác định độ dài của danh sách

int Length(***List*** *L);

3. Loại phần tử ở vị trí thứ k của danh sách

void Delete(***int*** k, ***List*** *L);

4. Xen phần tử X vào danh sách sau vị trí thứ k

void Insert_After(***Item*** X, ***int*** k, ***List*** *L);

5. Xen phần tử X vào danh sách trước vị trí thứ k

void Insert_Before(***Item*** X, ***int*** k, ***List*** *L);

6. Tìm phần tử X trong danh danh sách

int Search(***Item*** X, ***List*** *L);

Hàm Search trả về 1 nếu X có trong L, ngược lại trả về 0

7. Kiểm tra xem danh sách có rỗng không?

int Empty(***List*** *L); *//Hàm Empty trả về 1 nếu L rỗng, ngược lại trả về 0*

8. Kiểm tra xem danh sách có đầy không?

int Full(***List*** *L); *//Hàm Full trả về 1 nếu L đầy, ngược lại trả về 0*

9. Duyệt danh sách

Trong nhiều ứng dụng chúng ta phải đi qua danh sách, từ đầu đến cuối danh sách và thực hiện một nhóm các thao tác nào đó đối với mỗi phần tử của danh sách.

void Traverse(***List*** *L);

10. Các phép toán khác

Còn có thể kể ra nhiều phép toán khác. Chẳng hạn truy nhập đến phần tử thứ i của danh sách (để tham khảo hoặc thay thế), kết hợp hai danh sách thành một danh sách, tách một danh sách thành nhiều danh sách, v.v...

Ví dụ: Giả sử có danh sách L = (3, 2, 1, 5). Khi đó, thực hiện Delete(3, L) ta được danh sách (3, 2, 5). Kết quả của Insert_Before(1, 6, L) ta được danh sách (6, 3, 2, 1, 5).

Sau đây ta sẽ xét một số loại danh sách và ứng dụng của chúng.

2. LƯU TRỮ KẾ TIẾP CỦA DANH SÁCH TUYẾN TÍNH

Ta biết rằng danh sách tuyến tính là một danh sách hoặc rỗng, hoặc có dạng L = (a1, a2,..., an). Trong danh sách tuyến tính luôn tồn tại một phần tử đầu là **a1** và một phần tử cuối là **an** ($n \geq 1$).

Để lưu trữ danh sách tuyến tính trong bộ nhớ máy tính, một phương pháp rất tự nhiên là sử dụng mảng một chiều, trong đó mỗi thành phần của mảng lưu trữ một phần tử tương ứng của danh sách, các phần tử kế nhau của danh sách được lưu trữ trong các thành phần kế nhau của mảng. Lưu trữ danh sách theo cách này gọi là lưu trữ kế tiếp.

Tuy nhiên, việc sử dụng mảng một chiều cũng có những ưu điểm và nhược điểm nhất định của nó:

+ Vì mảng được lưu trữ kế tiếp nên việc truy nhập vào một thành phần nào đó được thực hiện trực tiếp dựa vào địa chỉ tính được (chỉ số), nên tốc độ nhanh và đồng đều đối với mọi phần tử.

+ Khi khai báo một mảng ta phải xác định số lượng phần tử của mảng, điều này sẽ tuỳ thuộc vào số lượng phần tử của danh sách mà mảng sẽ lưu trữ, nhưng điều này rất khó thực hiện vì số lượng phần tư của danh sách luôn luôn biến động. Do đó, có thể dẫn đến lãng phí bộ nhớ (có những phần tử mảng không được sử dụng) hoặc thiếu bộ nhớ (do tất cả các phần tử mảng đã được sử dụng trong khi ta cần thêm vào danh sách một số phần tử nào đó).

Sau đây ta trình bày cách cài đặt danh sách tuyến tính bởi mảng một chiều:

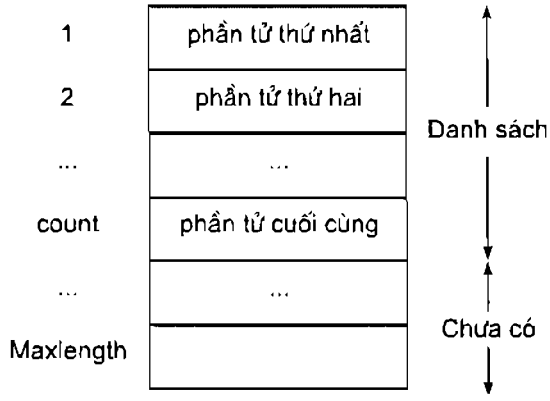
Giả sử độ dài tối đa của danh sách là một số nguyên dương N nào đó, các phần tử trong danh sách có kiểu dữ liệu là ***Item***. ***Item*** có thể là các kiểu dữ liệu đơn (số nguyên, số thực, ký tự), hoặc các kiểu dữ liệu có cấu trúc (chuỗi, cấu trúc). Danh sách được biểu diễn bởi một cấu trúc gồm hai thành phần dữ liệu.

+ Thành phần thứ nhất là mảng các ***Item***, phần tử thứ i của danh sách được lưu trữ bởi phần tử thứ i của mảng.

+ Thành phần thứ hai ghi chỉ số của phần tử mảng lưu trữ phần tử cuối cùng của danh sách.

Ta có khai báo cấu trúc dữ liệu của danh sách như sau:

```
#define Maxlength = N
// Khai báo kiểu Item (nếu cần)
struct List
{
        Item E[Maxlength];
        int count;
};
struct List L; //Khai báo danh sách L
```



Hình 4.1: Mảng biểu diễn danh sách

Trong cách cài đặt danh sách bởi mảng, các phép toán trên danh sách được thực hiện rất dễ dàng. Để khởi tạo danh sách rỗng chỉ cần một lệnh gán:

L.count = 0;

Độ dài thực của danh sách là L.count, danh sách đầy nếu L.count=Maxlength.

Ví dụ: Khai báo danh sách lưu trữ thông tin về sinh viên

```
#define Maxlength 100
struct student                  //Item ở đây là student
{
  char std_no[10];              //Mã sinh viên
  char std_name[30];            //Họ tên
  int age;                      //Tuổi
  float avg_point;              //Điểm trung bình
};
struct List
{
  struct student E[Maxlength];
  int count;
};
struct List L;
```

Dưới đây ta cài đặt hai phép toán trên danh sách: phép toán bổ sung một phần tử mới vào danh sách và phép toán loại bỏ một phần tử khỏi danh sách.

1. Loại bỏ một phần tử ở vị trí k trong danh sách

```
int DeleteL(int k, struct List *L)
{
  int i;
  if (k>=1 && k <= L->count)
  {
        i = k;
        while (i < L->count)
        {
              L->E[i] = L->E[i+1] ;
              i = i + 1;
        }
        L->count = L->count - 1;
        return 1;
  }
  else return 0;
}
```

Hàm DeleteL thực hiện phép loại một phần tử ở vị trí k trong danh sách. Phép toán được thực hiện khi danh sách không rỗng và k chỉ vào một phần tử trong danh sách. Giá trị trả về của hàm cho biết phép toán có được thực hiện thành công hay không (trả về 1 nếu thành công, trả về 0 nếu không thành công). Khi loại bỏ, ta phải dồn các phần tử ở các vị trí k+1, k+2,..., L.count lên trên một vị trí và giảm số lượng phần tử của danh sách đi một đơn vị (L.count = L.count – 1).

2. Bổ sung một phần tử vào trước phần tử ở vị trí k trong danh sách (dữ liệu của phần tử này được lưu trong biến X).

```
int InsertL(int k, Item X, struct List *L)
{
  int i;
```

```
if (L->count < Maxlength && k <= L->count && k>=1)
{
        i = L->count + 1;
        while (i > k)
        {
                L->E[i] = L->E[i-1] ;
                i = i - 1;
        }
        L->count = L->count + 1;
        L->E[k] = X;
        return 1;
}
else return 0;
}
```

Hàm InsertL thực hiện phép bổ sung một phần tử vào trước phần tử ở vị trí k trong danh sách. Phép toán được thực hiện khi danh sách chưa đầy và k chỉ vào một phần tử trong danh sách. Giá trị trả về của hàm cho biết phép toán có được thực hiện thành công hay không (trả về 1: thành công, trả về 0: không thành công). Khi bổ sung, ta phải dãn các phần tử ở các vị trí L.count,..., k+1, k xuống dưới một vị trí và tăng số lượng phần tử của danh sách lên một đơn vị (L.count = L.count + 1).

**** Nhận xét về phương pháp cài đặt danh sách bởi mảng:***

Việc cài đặt danh sách bởi mảng có một số ưu điểm và nhược điểm sau:

Ưu điểm: Do tính chất của mảng, nên việc cài đặt danh sách bởi mảng cho phép ta truy nhập trực tiếp vào bất kỳ phần tử nào trong danh sách nên tốc độ truy nhập nhanh và đồng đều đối với mọi phần tử. Các phép toán cũng đều được thực hiện một cách dễ dàng.

Nhược điểm: Khi thực hiện các phép toán bổ sung một phần tử vào danh sách hoặc loại bỏ một phần tử ra khỏi danh sách ở vị trí trí k

nào đó, ta phải đẩy tất cả các phần tử sau k xuống dưới hoặc lên trên một vị trí, nên tốn nhiều thời gian. Tuy nhiên, nhược điểm chủ yếu của phương pháp cài đặt này là không gian nhớ cố định dành để lưu trữ các phần tử của danh sách. Không gian nhớ này bị quy định bởi kích thước của mảng (kích thước của mảng được xác định khi khai báo và nó không thể thay đổi trong khi thực hiện chương trình). Do đó có thể dẫn đến trường hợp lãng phí bộ nhớ (do khai báo kích thước mảng quá lớn so với số lượng các phần tử của danh sách) hoặc thiếu bộ nhớ (mảng đã đầy trong khi ta muốn bổ sung thêm một số phần tử nào đó vào danh sách).

Để khắc phục các nhược điểm trên đây người ta sử dụng một phương pháp khác để cài đặt danh sách tuyến tính đó là danh sách móc nối.

3. DANH SÁCH MÓC NỐI

Như đã nêu ở phần trên, lưu trữ kế tiếp đối với danh sách tuyến tính đã bộc lộ rõ nhược điểm trong trường hợp thực hiện thường xuyên các phép bổ sung hoặc loại bỏ phần tử, trường hợp xử lý đồng thời nhiều danh sách, v.v...

Việc sử dụng con trỏ hoặc mối nối để tổ chức danh sách tuyến tính, mà ta gọi là danh sách móc nối (hay còn gọi là danh sách liên kết), chính là một giải pháp nhằm khắc phục nhược điểm trên. Tuy nhiên, trước khi tìm hiểu về danh sách móc nối ta nhắc lại một số khái niệm về con trỏ, phương tiện được sử dụng để cài đặt danh sách móc nối.

3.1. Kiểu con trỏ và các khái niệm liên quan

Tất cả các biến có kiểu dữ liệu mà ta đã nghiên cứu như số, ký tự, mảng, cấu trúc..., được gọi là biến tĩnh vì chúng được xác định một cách rõ ràng khi khai báo, sau đó chúng được dùng thông qua tên. Thời gian tồn tại của biến tĩnh cũng là thời gian tồn tại của khối chương trình có chứa khai báo biến này. Chẳng hạn, các biến tĩnh được khai báo trong chương trình (biến toàn cục) sẽ tồn tại từ khi

chương trình được thực hiện cho đến khi kết thúc chương trình, còn các biến tĩnh được khai báo trong một hàm (biến cục bộ) sẽ tồn tại từ khi hàm được triệu gọi cho đến khi kết thúc.

Ngoài các biến tĩnh được xác định trước, người ta còn có thể tạo ra các biến trong lúc chạy chương trình, tuỳ theo nhu cầu. Việc tạo ra các biến theo kiểu này được gọi là cấp phát bộ nhớ động, các biến được tạo ra được gọi là biến động.

Các biến động không có tên. Trong C/C++, để tạo ra biến động, người ta sử dụng một kiểu biến đặc biệt, gọi là con trỏ và các hàm/toán tử cấp phát bộ nhớ động (*malloc(), calloc(), realloc() trong thư viện malloc.h, toán tử new*) thông qua con trỏ. Khi không sử dụng biến động nữa, người ta có thể xoá nó khỏi bộ nhớ, việc này gọi là thu hồi bộ nhớ động. Để thu hồi bộ nhớ dành cho biến động, người ta dùng hàm *free()*/toán tử *delete* và thông qua con trỏ đã sử dụng để tạo ra biến động.

So với biến tĩnh, việc sử dụng biến động có ưu điểm là tiết kiệm được bộ nhớ. Bởi vì, khi cần dùng biến động thì người ta sẽ tạo ra nó và khi không cần nữa người ta lại có thể xoá nó khỏi bộ nhớ. Còn đối với các biến tĩnh, chúng được xác định và cấp phát bộ nhớ khi biên dịch, chúng sẽ chiếm giữ bộ nhớ trong suốt thời gian chương trình làm việc. Chẳng hạn, nếu cần sử dụng một mảng ta phải khai báo ngay ở phần đầu chương trình, ngay lúc này ta đã phải xác định kích thước của mảng và thường khai báo dôi ra, gây lãng phí bộ nhớ.

3.1.1. Con trỏ

Con trỏ là một biến dùng để chứa địa chỗ nhớ chỉ của một biến khác.

Cách khai báo con trỏ

<Kiểu dữ liệu> <*tên con trỏ>;

Ví dụ:

int *p, *q; //Khai báo p và q là hai con trỏ kiểu nguyên

Nghĩa là p, q được dùng để lưu địa chỉ của các biến nguyên.

3.1.2. Các phép toán con trỏ

Giả sử có khai báo ***int*** *p, *q, x;

Khi đó ta có thể thực hiện các phép toán

+ Gán địa chỉ cho con trỏ

Ví dụ: p = &x; //Gán địa chỉ của biến x cho p, hay p trỏ vào x.

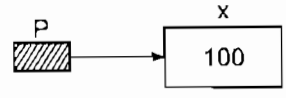
+ Phép gán hai con trỏ cùng kiểu. *Ví dụ*: q = p; //q và p cùng trỏ vào x.

+ Phép so sánh hai con trỏ cùng kiểu gồm: so sánh == (bằng nhau) và phép sánh != (khác nhau).

Phép toán một ngôi * được sử dụng với con trỏ để trả về giá trị của chỗ nhớ do con trỏ trỏ đến, hoặc làm thay đổi giá trị của chỗ nhớ đó.

Cách viết <***tên_con_trỏ***>

Ví dụ:




```
int *p, x, y;
x = 100;
p = &x;          //p trỏ vào x
y = *p;          //khi đó ta có giá trị của y = 100
*p = 500;        //Khi đó ta cũng có x = 500
```

3.1.3. Giá trị NULL

NULL là một giá trị con trỏ đặc biệt dành cho các biến con trỏ, nó được dùng để báo rằng con trỏ không lưu địa chỉ của biến nào. Giá trị NULL có thể được đem gán cho bất kỳ biến con trỏ nào. Đương nhiên khi đó việc thâm nhập vào biến động thông qua con trỏ có giá trị NULL là vô nghĩa.

3.1.4. Con trỏ cấu trúc

Con trỏ chứa địa chỉ của một biến cấu trúc được gọi là con trỏ cấu trúc, khi đó ta có thể thao tác với cấu trúc thông qua con trỏ. Việc truy nhập vào các thành phần của cấu trúc bằng con trỏ được viết theo cách sau:

<tên_con_trỏ> —> <tên thành phần>

Ví dụ:

```
struct   Hoc_sinh
{
         char   Ho_ten[25];
         int    tuoi;
         float  diem;
};
struct Hoc_sinh *p, h;
p = &h;
```

Khi đó việc truy xuất vào các thành phần của cấu trúc h thông qua con trỏ p được viết như sau:

```
strcpy(p->Ho_ten, "Nguyen Trong Huan");
p->diem = 7.4;
cin>>p->tuoi;
```

3.1.6. Cấp phát và thu hồi bộ nhớ động

Trong ngôn ngữ lập trình C, có thể sử dụng các hàm cấp phát bộ nhớ động gồm: malloc(), calloc(), các hàm này được định nghĩa trong thư viện malloc.h

+ Hàm malloc(), cấp phát cho con trỏ một vùng nhớ liên tiếp, kích thước vùng nhớ được chỉ ra bởi tham số size.

Cú pháp: ***void *malloc(size_t size)***

Trong đó size là kích thước vùng nhớ được cấp phát cho con trỏ được tính bằng byte.

Ví dụ:

```
int *p;
p=(int*) malloc (sizeof(int));
```

Câu lệnh trên cấp phát cho con trỏ p một chỗ nhớ có kích thước bằng một dữ liệu kiểu ***int***.

sizeof là toán tử trả về kích thước chỗ nhớ của một dữ liệu thuộc một kiểu dữ liệu nào đó.

+ Hàm calloc(), cấp phát một vùng nhớ gồm n chỗ nhớ.

Cú pháp: ***void** *calloc(**int** n, **size_t** size)*

Ví dụ:

```
int *p;
p=(int*)calloc(1, sizeof(int);
```

Dưới đây ta xét một chương trình được cài đặt với con trỏ cấu trúc:

```
#include<stdio.h>
#include<conio.h>
#include<malloc.h>
struct   Hoc_sinh
{
        char   ht[25];
        int    tuoi;
        char   qq[40];
};
void Nhap_du_lieu(struct Hoc_sinh *p)
{
        cout<< "\tHo ten: " ; fflush(stdin); gets(p->ht);
        cout<< "\tTuoi: "; cin>>p->tuoi;
        cout<< "\tQue quan: "; fflush(stdin) ; gets(p->qq);
}
void Hien_thi(struct Hoc_sinh p)
{       cout<< "\tHo ten: "<<p->ht<<endl;
        cout<< "\tTuoi: "<<p->tuoi<<endl;
        cout<< "\tQue quan: "<<p->qq<<endl;
}
void main()
{
        struct Hoc_sinh *p;
        p=(struct Hoc_sinh*) malloc(sizeof(struct Hoc_sinh));
```

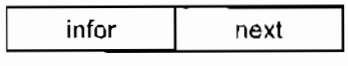
```
        cout<< "Nhap thong tin ve hoc sinh"<<endl;
        Nhap_du_lieu(p);
        cout<< "Thong tin hoc sinh vua nhap"<<endl;
        Hien_thi(*p);
        free(p);
        getch();
}
```

Qua ví dụ trên ta thấy rằng, để đưa dữ liệu vào bộ nhớ thông qua biến con trỏ ta cần phải cấp phát bộ nhớ cho nó và sau khi không sử dụng nữa ta có thể xoá nó khỏi bộ nhớ. Tuy nhiên, nếu chỉ lưu trữ dữ liệu đơn giản như vậy thì ta không cần đến biến con trỏ, nó được sử dụng trong một ứng dụng quan trọng hơn đó là việc cài đặt danh sách liên kết. Sau đây ta xét tới một số dạng danh sách móc nối.

3.2. Danh sách móc nối đơn

3.2.1. Nguyên tắc

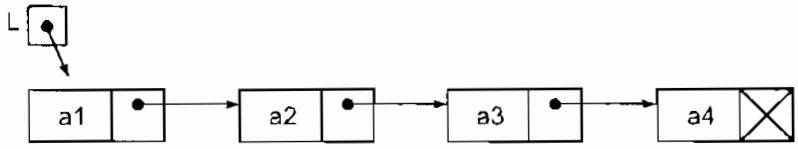
Trong cách cài đặt này, danh sách móc nối được tạo nên từ các phần tử nhỏ mà ta gọi là nút (Node). Các nút này có thể nằm bất kỳ đâu trong bộ nhớ máy tính. Mỗi nút là một cấu trúc gồm hai thành phần, *infor* chứa thông tin của phần tử trong danh sách, *next* là một con trỏ, nó trỏ vào nút đứng sau. Qui cách của mỗi nút có thể hình dung như sau:



Riêng nút cuối cùng thì không có nút đứng sau nó nên thành phần *next* của nút này có giá trị NULL để báo kết thúc danh sách.

Để có thể truy nhập vào mọi nút trong danh sách, ta phải truy nhập từ nút đầu tiên, nghĩa là cần có một con trỏ L trỏ tới nút đầu tiên này.

Nếu dùng mũi tên để chỉ mối nối, ta sẽ có hình ảnh của một danh sách móc nối đơn như hình 4.2:



Hình 4.2: Biểu diễn danh sách móc nối đơn

Dấu ⊠ chỉ giá trị trường *next* của phần tử này bằng NULL.

Giả sử các phần tử trong danh sách có kiểu dữ liệu là ***Item***. Dưới đây là khai báo cấu trúc dữ liệu biểu diễn danh sách móc nối đơn.

```
//định nghĩa kiểu dữ liệu Item (nếu cần)
struct Node
{
        Item infor;
        struct Node *next;
};
struct Node *L;  //Khai báo con trỏ L trỏ vào đầu danh sách
```

L = NULL nếu danh sách rỗng.

Ví dụ: Khai báo danh sách lưu trữ thông tin về sinh viên:

```
struct student                  //Item ở đây là student
{
        char std_no[10];        //Mã sinh viên
        char std_name[30];      //Họ tên
        int age;                //Tuổi
        float avg_point;        //Điểm trung bình
};
struct Node
{
        struct student infor;
        struct Node *next;
};
struct Node *L;  //Khai báo biến con trỏ L trỏ vào đầu danh sách
```

3.2.2. Các phép toán trên danh sách móc nối đơn

Bây giờ chúng ta sẽ xem xét một số phép toán tác động trên danh sách nối đơn.

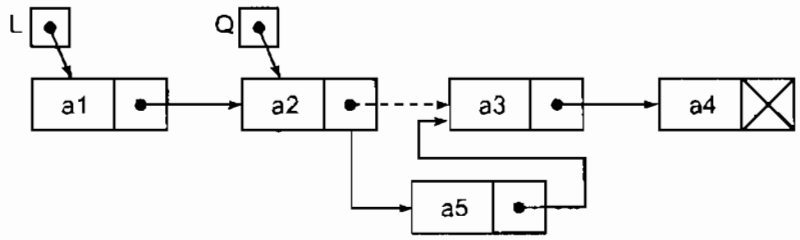
Điều kiện để danh sách móc nối đơn rỗng là L = NULL. Do đó, để khởi tạo danh sách rỗng ta chỉ cần lệnh gán: L = NULL;

Danh sách móc nối chỉ đầy khi không còn không gian nhớ để cấp phát cho các phần tử mới của danh sách. Chúng ta giả thiết điều này không xảy ra, nghĩa là danh sách móc nối không bao giờ đầy. Do đó, phép toán bổ sung một phần tử vào danh sách luôn luôn được thực hiện.

a. Bổ sung một nút mới vào danh sách móc nối đơn

Giả sử Q là một con trỏ, trỏ vào một nút trong danh sách, ta cần bổ sung một phần tử mới với thông tin lưu trong biến X vào sau nút được trỏ bởi Q. Phép toán này được thực hiện bởi thủ tục sau:

```
void InsertAfter(struct Node **L, struct Node *Q , Item X)
{
  struct Node *p;
//1. Tạo một nút mới
        p=(struct Node *)malloc(sizeof(struct Node));
        p->infor = X;
//2. Thực hiện bổ sung, nếu danh sách rỗng thì bổ sung nút mới vào thành nút đầu tiên, ngược lại bổ sung nút mới vào sau nút được trỏ bởi Q.
        if (*L == NULL)
        {
                p->next = NULL;
                *L = p;
        }
        else {
                p->next = Q->next;
                Q->next = p;
        }
}
```



Hình 4.3: Mô tả phép bổ sung một phần tử vào sau nút trỏ bởi Q trong danh sách

Giả sử bây giờ ta cần bổ sung nút mới vào trước nút được trỏ bởi Q. Phép toán này phức tạp hơn. Khó khăn là ở chỗ, nếu Q không phải là nút đầu tiên của danh sách (Q≠L) thì ta không thể xác định được nút đứng trước Q để kết nối nó với nút mới. Có thể giải quyết khó khăn bằng cách, đầu tiên ta vẫn bổ sung nút mới vào sau Q, sau đó trao đổi giá trị chứa trong phần infor giữa nút mới và nút được trỏ bởi Q. Thủ tục thực hiện phép toán này xin dành cho bạn đọc.

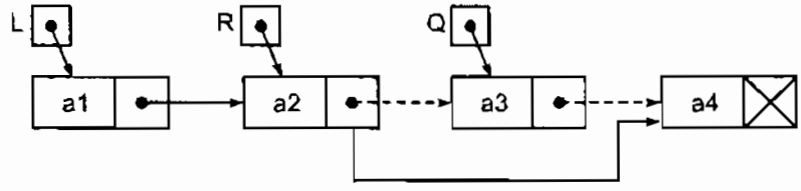
b. Loại bỏ một nút ra khỏi danh sách móc nối đơn

Cho danh sách móc nối đơn được trỏ bởi L. Q là một con trỏ, trỏ vào một nút trong danh sách. Giả sử ta cần loại bỏ nút được trỏ bởi Q. Ở đây ta cũng gặp khó khăn là nếu Q không phải là nút đầu tiên thì không xác định được nút đứng trước Q. Trong trường hợp này ta phải tìm đến nút đứng trước Q và cho con trỏ R trỏ vào nút đó, tức là Q = R->next. Sau đó ta mới thực hiện loại bỏ nút Q. Ta có thủ tục sau:

```
int DeleteL(struct Node **L, struct Node *Q, Item &X)
{
  struct Node *R;
  //1. Trường hợp danh sách rỗng
  if (*L = = NULL)
  {
        return 0;
  }
  X = Q->infor;   //lưu thông tin của nút cần loại bỏ vào biến X
```

```
    //2. Trường hợp nút trỏ bởi Q là nút đầu tiên
        if (Q == *L)
        {
            *L = Q->next; free(Q);
            return 1;
        }
    //3. Tìm đến nút đứng trước nút trỏ bởi Q
        R = *L;
        while (R->next != Q)
            R = R->next;
    //4. Loại bỏ nút trỏ bởi Q
        R->next = Q->next;     free(Q);
}
```

Phép toán loại bỏ được mô tả bởi hình 4.4.



Hình 4.4: Mô tả phép loại bỏ một phần tử ra khỏi danh sách

c. Ghép hai danh sách móc nối đơn

Giả sử có hai danh sách móc nối đơn lần lượt được trỏ bởi L1 và L2. Thủ tục sau thực hiện việc ghép hai danh sách đó thành một danh sách mới được trỏ bởi L1.

```
void COMBINE(struct Node **L1, struct Node *L2)
{
  struct Node *R;
  //1. Trường hợp danh sách trỏ bởi L2 rỗng
  if (L2 == NULL) return;
  //2. Trường hợp danh sách trỏ bởi P rỗng
  if (*L1 == NULL)
```

```
{
        *L1 = L2; return;
}
//3. Tìm đến nút cuối danh sách trỏ bởi P
R = *L1;
while (R->next != NULL)
        R = R->next;
//4. Ghép
R->next = L2;
}
```

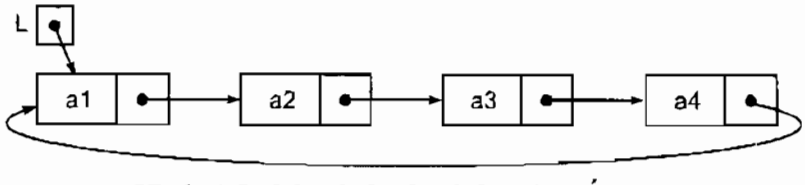
Nhận xét: Rõ ràng với các danh sách tuyến tính mà kích thước luôn biến động trong quá trình xử lý hay thường xuyên có các phép bổ sung và loại bỏ tác động, thì việc lưu trữ bằng danh sách móc nối như trên tỏ ra thích hợp. Tuy nhiên, cách cài đặt này cũng có những nhược điểm nhất định:

Chỉ có phần tử đầu tiên trong danh sách được truy nhập trực tiếp, các phần tử khác chỉ được truy nhập sau khi đã đi qua các phần tử đứng trước nó.

Ở mỗi nút trong danh sách phải có thêm trường *next* để lưu trữ địa chỉ của nút tiếp theo, do đó với cùng một danh sách thì việc cài đặt bởi danh sách móc nối sẽ tốn bộ nhớ hơn so với cài đặt bằng mảng.

3.3. Danh sách nối vòng

Một cải tiến của danh sách móc nối đơn là kiểu danh sách móc nối vòng. Nó khác với danh sách móc nối đơn ở chỗ: trường *next* của nút cuối cùng trong danh sách không phải bằng NULL, mà nó trỏ đến nút đầu tiên trong danh sách, tạo thành một vòng tròn. Hình ảnh của nó như hình 4.5.

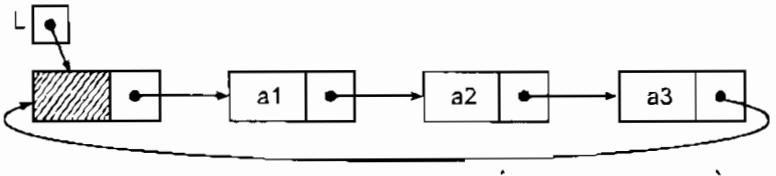


Hình 4.5: Mô tả danh sách móc nối vòng

Cải tiến này làm cho việc truy nhập vào các nút trong danh sách được linh hoạt hơn. Ta có thể truy nhập vào mọi nút trong danh sách bắt đầu từ nút nào cũng được, không nhất thiết phải từ nút đầu tiên. Điều đó có nghĩa là nút nào cũng có thể coi là nút đầu tiên và con trỏ L trỏ tới nút nào cũng được. Như vậy, đối với danh sách móc nối vòng chỉ cần cho biết con trỏ trỏ tới nút muốn loại bỏ ta sẽ thực hiện được vì luôn tìm được đến nút đứng trước đó. Với phép ghép, phép tách cũng có những thuận lợi nhất định.

Tuy nhiên, danh sách nối vòng có một nhược điểm rất rõ là trong khi xử lý, nếu không cẩn thận sẽ dẫn tới một chu trình không kết thúc, bởi vì không biết được vị trí kết thúc danh sách.

Để khắc phục nhược điểm này, người ta đưa thêm vào danh sách một nút đặc biệt gọi là "nút đầu danh sách". Trường infor của nút này không chứa dữ liệu của phần tử nào và con trỏ L bây giờ trỏ tới nút đầu danh sách này. Việc dùng thêm nút đầu danh sách đã khiến cho danh sách về mặt hình thức không bao giờ rỗng. Hình ảnh của nó minh họa như hình 4.6.



Hình 4.6: Mô tả danh sách móc nối vòng có nút đầu

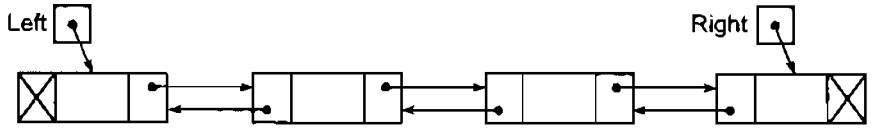
Sau đây là đoạn giải thuật bổ sung một nút vào thành nút đầu tiên trong danh sách có "nút đầu danh sách" trỏ bởi L.

```
P=(struct Node*) malloc(sizeof(struct Node));
P->infor = X;
P->next = L->next;
L->next = P;
```

3.4. Danh sách móc nối hai chiều

Khi làm việc với danh sách, có những xử lý trên mỗi nút của danh sách lại liên quan đến cả nút đứng trước và nút đứng sau. Trong

những trường hợp như thế, để thuận tiện, người ta đưa vào mỗi nút của danh sách hai con trỏ: *Next_Left* trỏ đến nút đứng trước và *Next_Right* trỏ đến nút đứng sau nó. Để truy nhập vào danh sách ta dùng hai con trỏ: con trỏ *Left* trỏ vào nút đầu tiên và con trỏ *Right* trỏ vào nút cuối cùng của danh sách. Hình ảnh của danh sách móc nối hai chiều được minh họa trên hình 4.7.



Hình 4.7: Mô tả danh sách móc nối đôi

Ta có thể khai báo cấu trúc dữ liệu danh sách móc nối hai chiều như sau:

```
//Khai báo kiểu dữ liệu Item (nếu cần)
struct Node
{
        Item infor;
        struct Node *Next_Left, *Next_Right;
};
struct Node *Left, *Right;
```

Việc cài đặt danh sách móc nối hai chiều sẽ tiêu tốn nhiều bộ nhớ hơn so với danh sách móc nối đơn. Song bù lại, danh sách móc nối đôi có những ưu điểm mà danh sách móc nối đơn không thể có được, chẳng hạn: khi xem xét danh sách móc nối đôi ta có thể lùi lại sau, hoặc tiến lên trước.

Các phép toán trên danh sách móc nối hai chiều được thực hiện dễ dàng hơn. Chẳng hạn, khi thực hiện phép toán loại bỏ, với danh sách móc nối đơn, ta không thể thực hiện được nếu không biết nút đứng trước nút cần loại bỏ. Trong khi đó, ta có thể tiến hành dễ dàng trên danh sách móc nối hai chiều.

Dưới đây là một số giải thuật tác động lên danh sách móc nối hai chiều nói trên.

3.4.1. Phép bổ sung một nút mới

Cho hai con trỏ Left và Right lần lượt trỏ tới nút đầu và nút cuối của một danh sách móc nối hai chiều, M là con trỏ trỏ tới một nút trong danh sách này. Giải thuật này thực hiện bổ sung một nút mới, mà dữ liệu chứa ở biến X, vào trước nút trỏ bởi con trỏ M.

```
void Bo_sung(struct Node **Left, struct Node **Right, struct Node
                *M, Item X)
{
  struct Node *P;
  //1. Tạo nút mới
  P = (Node*)malloc(sizeof(Node));
  P->infor = X;
  //Trường hợp danh sách rỗng
  if (*Right == NULL)
  {
        P->Next_Left = NULL; P->Next_Right = NULL;
        *Left = P; *Right = P;
  }
  else
  //Trường hợp M trỏ tới nút đầu tiên
  if (M == *Left)
  {     P->Next_Left = NULL;
        P->Next_Right = M
        M->Next_Left = P;
        *Left = P;
  }
  else //Bổ sung vào giữa
  {     P->Next_Left = M->Next_Left;
        P->Next_Right = M;
        M->Next_Left = P;
        P->Next_Left->Next_Right = P;
        }
}
```

3.4.2. Loại bỏ một nút trên danh sách

Cho hai con trỏ Left và Right lần lượt trỏ tới nút đầu và nút cuối của một danh sách móc nối hai chiều, M là con trỏ trỏ tới một nút trong danh sách này. Giải thuật này thực hiện loại bỏ nút trỏ bởi M ra khỏi danh sách.

```
int Loai_bo(struct Node **Left, struct Node **Right, struct Node *M)
{
  if (*Left = = NULL)
        return 0;
  else
        if (*Left = = *Right)
        {       *Left = NULL; *Right = NULL;
        }
        else
              if (M = = *Left)
              {       *Left = (*Left)->Next_Right;
                      (*Left)->Next_Left = NULL;
              }
              else if (M == Right)
              {       Right = Right^.Next_Left;
                      Right^.Next_Right = NULL;
              }
              else    {
                      M->Next_Left->Next_Right = M->Next_Right;
                      M->Next_Right->Next_Left = M->Next_Left;
                      }
}
```

Chú ý: Trong các ứng dụng, người ta cũng thường sử dụng các danh sách móc nối hai chiều vòng tròn, có nút đầu danh sách. Với loại danh sách này, ta có tất cả các ưu điểm của danh sách móc nối hai chiều và danh sách vòng tròn.

3.5. Ứng dụng danh sách móc nối: các phép tính số học trên đa thức

Trong mục này ta sẽ xét các phép tính số học cơ bản (cộng, trừ, nhân, chia) đối với đa thức một ẩn có dạng:

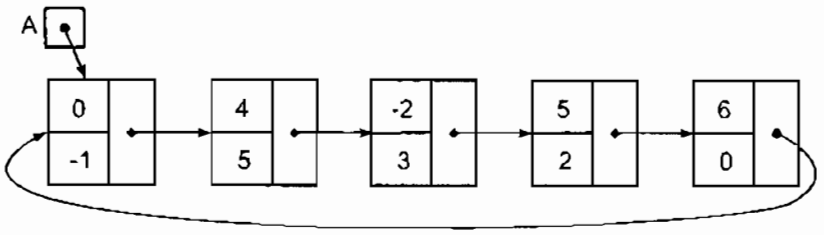
$$A(x) = a_n x^n + a_{n-1}x^{n-1} + ...+ a_1 x + a_0 \qquad (1)$$

Mỗi hạng thức của đa thức được đặc trưng bởi hệ số và số mũ của x. Giả sử các hạng thức trong đa thức được sắp xếp theo thứ tự giảm dần của số mũ, như trong đa thức (1). Ta thấy đa thức như một danh sách tuyến tính với các phần tử của danh sách là các hạng thức của đa thức. Khi ta thực hiện các phép toán trên đa thức ta sẽ nhận được đa thức có bậc không thể đoán trước được. Ngay cả những đa thức có bậc xác định thì số các hạng thức của nó cũng biến đổi rất nhiều từ một đa thức này đến một đa thức khác. Do đó, phương pháp tốt nhất là biểu diễn đa thức dưới dạng một danh sách móc nối. Mỗi nút của danh sách là một bản ghi gồm ba trường: coef chỉ hệ số, exp chỉ số mũ của x và con trỏ Next để trỏ tới nút tiếp theo. Cấu trúc dữ liệu mô tả một hạng thức (một nút) như sau:

```
struct Node
{
        float coef;
        int exp;
        struct Node *Next;
};
```

Vì những ưu điểm của danh sách vòng tròn có nút đầu danh sách (không cần kiểm tra danh sách rỗng, mọi thành phần đều có thành phần đi sau), ta chọn danh sách móc nối vòng tròn để biểu diễn đa thức. Với cách chọn này việc thực hiện các phép toán đa thức sẽ rất gọn. Nút đầu danh sách là nút đặc biệt, có exp = -1.

Như vậy với đa thức: $A(x) = 4x^5 - 2x^3 + 5x^2 + 6$ sẽ được biểu diễn như hình 4.8.



Hình 4.8: Danh sách móc nối đôi biểu diễn đa thức

Sau đây chúng ta sẽ xét phép cộng hai đa thức A(x) và B(x). Con trỏ A trỏ tới đầu danh sách biểu diễn đa thức A(x), con trỏ B trỏ tới đầu danh sách biểu diễn đa thức B(x). Sau khi thực hiện phép cộng hai đa thức trên ta được đa thức C(x) và con trỏ C trỏ tới đầu danh sách biểu diễn C(x).

**** Giải thuật***

Trước hết cần phải thấy rằng để thực hiện phép cộng đa thức A(x) với đa thức B(x) ta phải tìm đến từng hạng thức của các đa thức đó, nghĩa là phải dùng hai biến con trỏ P và Q để duyệt qua hai danh sách tương ứng với hai đa thức A(x) và B(x) trong quá trình tìm này.

Ta thấy có những trường hợp sau:

1. EXP(P) = EXP(Q), ta sẽ phải thực hiện cộng giá trị coef ở hai nút đó, nếu giá trị tổng khác không thì phải tạo ra nút mới thể hiện hạng thức tổng đó và gắn vào danh sách ứng với C(x).

2. EXP(P) > EXP(Q) (hoặc ngược lại cũng tương tự): phải sao chép nút P và gắn vào danh sách của C(x).

3. Nếu một danh sách kết thúc trước: phần còn lại của danh sách kia sẽ được sao chép và gắn vào danh sách của C(x).

Mỗi lần một nút mới được tạo ra đều phải gắn vào cuối danh sách C(x). Do đó, cần một con trỏ R trỏ vào nút cuối của danh sách C(x).

Công việc này được thực hiện nhiều lần, vì vậy cần được thể hiện bằng một thủ tục gọi là Attack(h, m, R). Nó thực hiện: lấy một nút mới, đưa vào trường coef của nút này giá trị h (hệ số), đưa vào

trường exp giá trị m (số mũ) và gắn nút mới đó vào sau nút trỏ bởi con trỏ R.

```
void Attack(float h, int m, struct Node *R)
{       struct Node *N;
        N=(struct Node*)malloc(sizeof(struct Node));
        N->coef = h;
        N->exp = m;
        R->Next = N;
        R = N;
}
```

Sau đây là thủ tục cộng hai đa thức:

```
void Add(struct Node *A, struct Node *B, struct Node **C)
{
        struct Node *P, *Q, *R;
        float x;
        P = A; Q = B;
        *C = (struct Node*)malloc(sizeof(struct Node));
        *C->exp = -1; R = *C;
        while (P->exp != -1 && Q->exp != -1)
        if (P->exp = Q->exp)
        {       x = P->coef + Q->coef;
                if (x != 0)
                        Attack(x, P->exp, R);
                P = P->Next; Q = Q->Next;
        }
        else    if (P->exp < Q->exp)
                {       Attack(Q->coef, Q->exp, R);
                        Q = Q->Next;
                }
                else {  Attack(P->coef, P->exp, R);
                }
  while (P->exp != -1)    //Danh sách ứng với B(x) đã hết
```

```
    {
            Attack(P->coef, P->exp, R);
            P = P->Next;
    }
    while (Q->exp != -1)    //Danh sách ứng với A(x) đã hết
    {
            Attack(Q->coef, Q->exp, R);
            Q = Q->Next;
    }
    R->Next = *C;
}
```

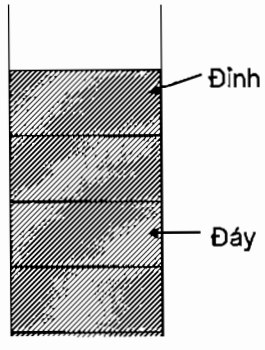
4. STACK VÀ QUEUE

4.1. Stack (Ngăn xếp)

4.1.1. Khái niệm

Ngăn xếp (Stack) là một danh sách tuyến tính, trong đó phép bổ sung một phần tử vào ngăn xếp và phép loại bỏ một phần tử khỏi ngăn xếp luôn luôn được thực hiện ở một đầu gọi là đỉnh (top).

Có thể hình dung Ngăn xếp như cơ cấu của một hộp tiếp đạn. Việc đưa đạn vào hộp đạn hay lấy đạn ra khỏi hộp chỉ được thực hiện ở đầu hộp. Viên đạn mới nạp nằm ở đỉnh còn viên đạn nạp đầu tiên nằm ở đáy hộp.



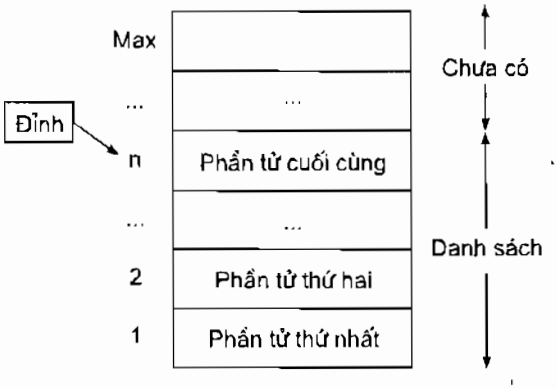
Hình 4.9: Ngăn xếp

4.1.2. Cài đặt ngăn xếp bởi mảng

Giả sử danh sách được biểu diễn là một ngăn xếp, có độ dài tối đa là một số nguyên dương N nào đó, các phần tử của ngăn xếp có kiểu dữ liệu là ***Item***. ***Item*** có thể là các kiểu dữ liệu đơn, hoặc các kiểu dữ liệu có cấu trúc. Chúng ta biểu diễn ngăn xếp bởi một bản ghi gồm 2 trường. Trường thứ nhất là mảng các ***Item***, trường thứ 2 ghi chỉ số của thành phần mảng lưu trữ phần tử ở đỉnh của ngăn xếp. Cấu trúc dữ liệu biểu diễn ngăn xếp được khai báo theo mẫu sau:

```
#define Max N
//Khai báo kiểu dữ liệu Item (nếu cần)
struct   Stack
{
        Item E[Max];
        unsigned int top;
};
struct Stack S; //Khai báo ngăn xếp S
```

Với cách cài đặt này, nếu S.top = 0 thì S là ngăn xếp rỗng, S.top = Max thì S là ngăn xếp đầy.



Hình 4.10: Mảng biểu diễn ngăn xếp

Ví dụ: Đoạn chương trình:

```
#define Max 100
struct   Hoc_sinh
```

```
{       char ho_ten[25];
        int tuoi;
};
struct Stack
{
        struct Hoc_sinh E[max];
        unsigned int top;
};
struct Stack S;
```

Khai báo ngăn xếp S có thể chứa tối đa 100 phần tử, mỗi phần tử (***Item***) là một cấu trúc Hoc_sinh gồm 2 thành phần ho_ten và tuoi.

Các phép toán trên ngăn xếp

Giả sử S là ngăn xếp, các phần tử của nó có kiểu ***Item*** và X là một phần tử có cùng kiểu với các phần tử của ngăn xếp. Ta có các phép toán sau với ngăn xếp S.

a. Khởi tạo ngăn xếp rỗng (ngăn xếp không chứa phần tử nào)

```
void Initialize (struct Stack *S)
{
        S->top = 0;
}
```

b. Kiểm tra ngăn xếp rỗng

```
int Empty (struct Stack S)
{
        return (S.top = = 0);
}
```

Hàm Empty nhận giá trị true nếu S rỗng và false nếu S không rỗng.

c. Kiểm tra ngăn xếp đầy

```
int Full (struct Stack S)
{
  return (S.top == Max);
}
```

Hàm *Full()* nhận giá trị true nếu S đầy và false nếu không.

d. Thêm một phần tử mới vào đỉnh ngăn xếp

Để bổ sung phần tử X vào đỉnh của ngăn xếp S, trước hết kiểm tra xem S có đầy không. Nếu S đầy thì bổ sung không thực hiện được, ngược lại X được bổ sung vào đỉnh của S. Hàm PUSH trả về 1 nếu bổ sung thành công, ngược lại trả về 0.

```
int PUSH (struct Stack *S, Item X)
{
        if (Full(S)) return 0;
        else
        {
                S->top = S->top + 1;
                S->E[S->top] = X;
                return 1;
        }
}
```

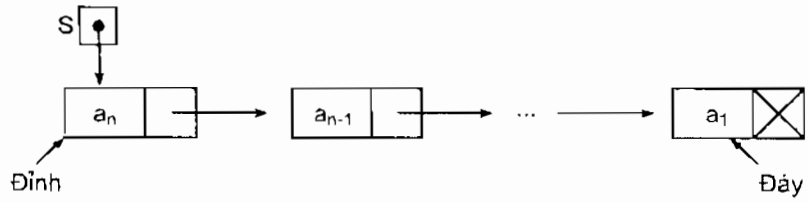
e. Loại bỏ phần tử ở đỉnh của ngăn xếp

Việc loại bỏ được thực hiện nếu S không rỗng, giá trị của phần tử bị loại bỏ được gán cho biến X. Hàm *POP*() trả về 1 nếu loại bỏ thành công, ngược lại trả về 0.

```
int POP (struct Stack *S; Item *X)
{
        if (Empty(*S)) return 0;
        else
        {       *X = S->E[S->top];
                S->top = S->top – 1;
                return 1;
        }
}
```

4.1.3. Cài đặt ngăn xếp bởi danh sách móc nối đơn

Để cài đặt ngăn xếp bởi danh sách móc nối đơn, ta sử dụng con trỏ S trỏ vào phần tử ở đỉnh của ngăn xếp (hình 4.11).



Hình 4.11: Danh sách móc nối đơn biểu diễn ngăn xếp

Cấu trúc dữ liệu của ngăn xếp được khai báo như sau:

```
struct Node
{
        Item Infor;
        Node *Next;
};
struct Node *S;
```

Trong cách cài đặt này, ngăn xếp rỗng khi S = NULL. Ta giả sử việc cấp phát bộ nhớ động cho các phần tử mới luôn thực hiện. Do đó, ngăn xếp không bao giờ đầy và phép toán PUSH luôn thực hiện thành công.

) Các hàm và thủ tục thực hiện các phép toán trên ngăn xếp:

a. Khởi tạo ngăn xếp rỗng:

```
void Create(struct Node **S)
{       *S = NULL;
}
```

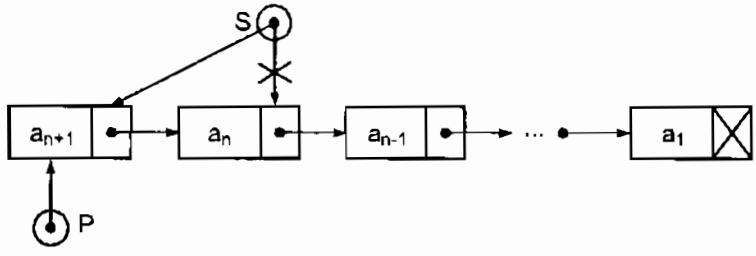
b. Kiểm tra ngăn xếp rỗng:

```
int Empty(struct Node *S)
{       return (S == NULL);
}
```

c. Bổ sung một phần tử vào đỉnh ngăn xếp:

```
void PUSH(struct Node **S, Item X)
{
        struct Node *P;
```

```
        P = new Node;
        P->Infor = X;
        P->Next = NULL;
        if (*S== NULL)   *S = P;
        else
        {       P->Next = *S; *S = P;
        }
}
```

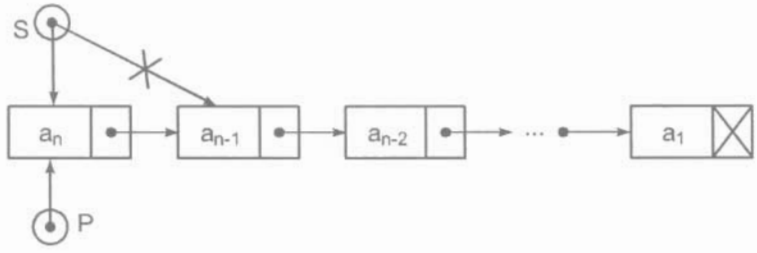


Hình 4.12: Minh hoạ thao tác PUSH

d. Lấy ra một phần tử ở đỉnh ngăn xếp:

```
int POP(struct Node **S, Item *X)
{
        struct Node *P;
        if (Empty(*S)) return 0;
        else
        {
                P = *S;
                X = (*S)->Infor;
                *S = (*S)->Next;
                delete P;
                return 1;
        }
}
```

Hàm POP trả về 1 nếu việc loại bỏ thành công, ngược lại trả về 0.

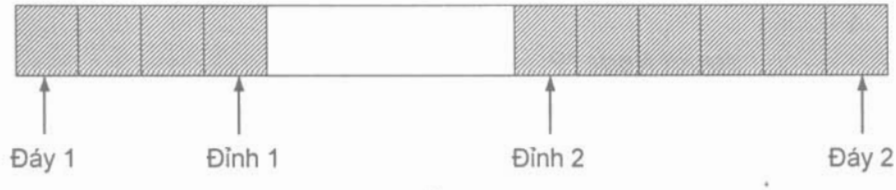


Hình 4.13: Minh hoạ thao tác POP

4.1.4. Xử lý với nhiều ngăn xếp

Có những trường hợp cùng một lúc ta phải xử lý nhiều ngăn xếp trên cùng một không gian nhớ. Như vậy, có thể xảy ra tình trạng một ngăn xếp này đã bị tràn trong khi không gian dự trữ cho ngăn xếp khác vẫn còn chỗ trống (tràn cục bộ). Làm thế nào để khắc phục được tình trạng này?

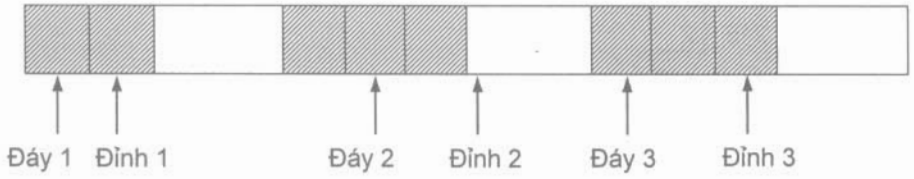
Nếu là hai ngăn xếp thì có thể giải quyết dễ dàng. Ta không qui định kích thước tối đa cho từng ngăn xếp nữa mà không gian nhớ dành ra sẽ được dùng chung. Ta đặt hai ngăn xếp ở hai đầu sao cho hướng phát triển của chúng ngược nhau, như hình 4.14.



Hình 4.14: Hai ngăn xếp trên một không gian nhớ

Như vậy, có thể một ngăn xếp này dùng lấn sang quá nửa không gian dự trữ nếu như ngăn xếp kia chưa dùng đến. Do đó hiện tượng tràn chỉ xảy ra khi toàn bộ không gian nhớ dành cho chúng đã được dùng hết.

Nhưng nếu số lượng ngăn xếp từ 3 trở lên thì không thể làm theo kiểu như vậy được, mà phải có giải pháp linh hoạt hơn nữa. Chẳng hạn có 3 ngăn xếp, lúc đầu không gian nhớ có thể chia đều cho cả 3, như hình 4.15.



Hình 4.15: Ba ngăn xếp trên một không gian nhớ

Nhưng nếu có một ngăn xếp nào phát triển nhanh bị tràn trước mà ngăn xếp khác vẫn còn chỗ thì phải dồn chỗ cho nó bằng cách hoặc đẩy ngăn xếp đứng sau nó sang bên phải hoặc lùi chính ngăn xếp đó sang trái trong trường hợp có thể. Như vậy thì đáy của các ngăn xếp phải được phép di động và dĩ nhiên các giải thuật bổ sung hoặc loại bỏ phần tử đối với các ngăn xếp hoạt động theo kiểu này cũng phải thay đổi.

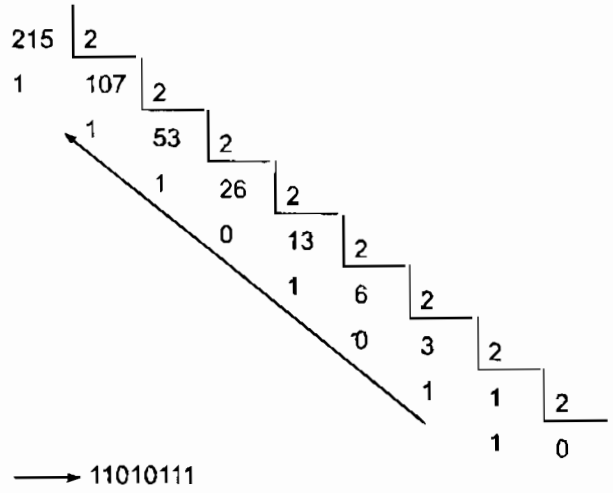
4.1.5. Một số ứng dụng của ngăn xếp

a. Ứng dụng đổi cơ số

Ta biết rằng dữ liệu lưu trữ trong bộ nhớ của máy tính đều được biểu diện dưới dạng mã nhị phân. Như vậy các số xuất hiện trong chương trình đều phải chuyển đổi từ hệ thập phân sang hệ nhị phân trước khi thực hiện các phép xử lý.

Khi đổi một số nguyên từ hệ thập phân sang hệ nhị phân người ta dùng phép chia liên tiếp cho 2 và lấy các số dư (là các chữ số nhị phân) theo chiều ngược lại.

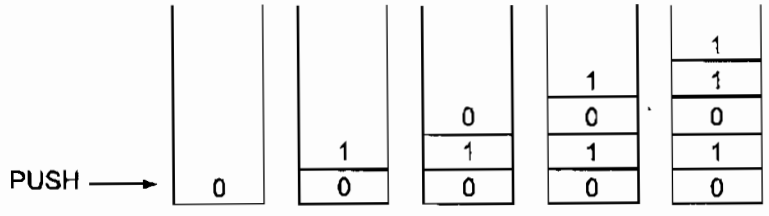
Ví dụ: Đổi số 215 sang hệ nhị phân:



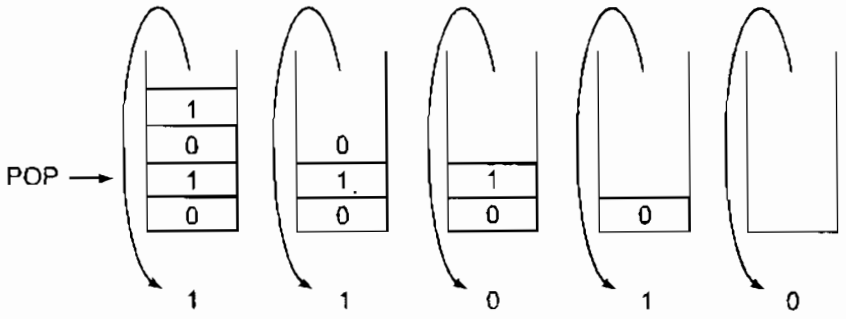
Ta thấy trong cách biến đổi này các số được tạo ra sau lại được hiển thị trước. Cơ chế sắp xếp này chính là cơ chế hoạt động của ngăn xếp. Để thực hiện biến đổi ta sẽ dùng một ngăn xếp để lưu trữ các số dư qua từng phép chia: Khi thực hiện phép chia thì nạp số dư vào ngăn xếp, sau đó lấy chúng lần lượt từ ngăn xếp ra.

Ví dụ:

Số: $(26)_{10} = (11010)_2$ trong quá trình biến đổi các số dư lần lượt sẽ là: (0 1 0 1 1).



Hình 4.16(a): Mô tả hoạt động lưu trữ dữ liệu vào ngăn xếp



Hình 4.16.(b): Mô tả hoạt động lấy dữ liệu từ ngăn xếp

Ta khai báo cấu trúc dữ liệu cho bài toán này như sau:

```
#define Max 16  //thực hiện đổi số nguyên có kích thước 2 byte
typedef int Item; //Item là chữ số nhị phân
struct    Stack
{
        Item E[Max];
        int top;
};
struct Stack S;
/*S là ngăn chứa các số dư qua các phép chia trong quá trình chuyển đổi*/
```

Giải thuật sử dụng ngăn xếp thực hiện chuyển đổi số nguyên dương N từ hệ cơ số 10 sang hệ cơ số 2. Trong giải thuật này có sử dụng các phép toán trên ngăn xếp.

```
void Chuyen_doi(N)
{
  1 - while (N != 0)
        R = N % 2;      //tính số dư trong phép chia N cho 2
        call PUSH(S, R);
        N = N/2;        //thay N bằng thương của phép chia N cho 2
  2 - cout<<"\nma nhi phan: ";
        while (!Empty(S))
```

```
        {
            call POP(S, R);
            cout<<R;
        }
}
```

b. Ứng dụng định giá biểu thức số học theo ký pháp nghịch đảo

Nhiệm vụ của bộ dịch là tạo ra các chỉ thị máy cần thiết để thực hiện các lệnh của chương trình nguồn. Một phần trong nhiệm vụ này là tạo ra các chỉ thị định giá các biểu thức số học. Chẳng hạn câu lệnh gán X = A*B + C.

Bộ dịch phải tạo ra các chỉ thị máy tương ứng như sau:

1 - LOA A: Tìm giá trị của A lưu trữ trong bộ nhớ và tải nó vào thanh ghi.

2 - MUL B: Tìm giá trị của B và nhân nó với giá trị đang ở thanh ghi.

3 - ADD C: Tìm giá trị của C và cộng nó với giá trị trong thanh ghi.

4 - STO X: Đưa giá trị trong thanh ghi vào lưu trữ ở vị trí tương ứng của X, trong bộ nhớ.

Trong các ngôn ngữ lập trình, biểu thức số học được viết như dạng thông thường của toán học nghĩa là theo kí pháp trung tố (infix notation) mỗi kí hiệu của phép toán hai ngôi được đặt giữa hai toán hạng, có thể thêm dấu ngoặc.

Chẳng hạn: 5*(7+3)

Dấu ngoặc là cần thiết vì nếu viết 5*7 + 3 thì theo qui ước về thứ tự ưu tiên của phép toán (mà các ngôn ngữ lập trình đều chấp nhận) thì biểu thức trên nghĩa là lấy 5 nhân 7 được kết quả cộng với 3.

Nhà logic học người Ba Lan Lukasiewicz đã đưa ra dạng biểu thức số học theo ký pháp hậu tố (postfix notation) và tiền tố (prefix notation) mà được gọi là dạng ký pháp Ba Lan.

Ở dạng hậu tố các toán tử đi sau các toán hạng. Như biểu thức 5*(7+3) sẽ có dạng: 5 7 3 - *

Còn ở dạng tiền tố thì các toán tử sẽ đi trước các toán hạng. Khi đó biểu thức 5*(7+3) có dạng: * 5 - 7 3

Ông cũng khẳng định rằng đối với các dạng ký pháp này dấu ngoặc là không cần thiết.

Nhiều bộ dịch khi định giá biểu thức số học thường thực hiện: trước hết chuyển các biểu thức dạng trung tố có dấu ngoặc sang dạng hậu tố, sau đó mới tạo các chỉ thị máy để định giá biểu thức ở dạng hậu tố. Việc biến đổi từ dạng trung tố sang dạng dạng hậu tố không khó khăn gì, còn việc định giá theo dạng hậu tố thì dễ dàng hơn, "máy móc" hơn so với dạng trung tố.

Để minh hoạ ta xét định giá của biểu thức sau:

1 5 + 8 4 1 - - *

tương ứng với biểu thức thông thường: (1 + 5) * (8 - (4 - 1))

Biểu thức này được đọc từ trái sang phải cho tới khi tìm ra một toán tử. Hai toán hạng được đọc cuối cùng, trước toán tử này, sẽ được kết hợp với nó. Trong ví dụ của chúng ta thì toán tử đầu tiên được đọc là + và hai toán hạng tương ứng với nó là 1 và 5, sau khi kết hợp biểu thực con này có giá trị là 6, thay vào ta có biểu thức rút gọn:

6 8 4 1 - - *

Lại đọc từ trái sang phải, toán tử tiếp theo là - và ta xác định được 2 toán hạng của nó là 4 và 1. Thực hiện phép toán ta có dạng rút gọn:

6 8 3 - *

Lại tiếp tục ta đi tới:

6 5 *

và cuối cùng thực hiện phép toán * ta có kết quả là 30.

Phương pháp định giá biểu thức hậu tố như trên đòi hỏi phải lưu trữ các toán hạng cho tới khi một toán tử được đọc, tại thời điểm này

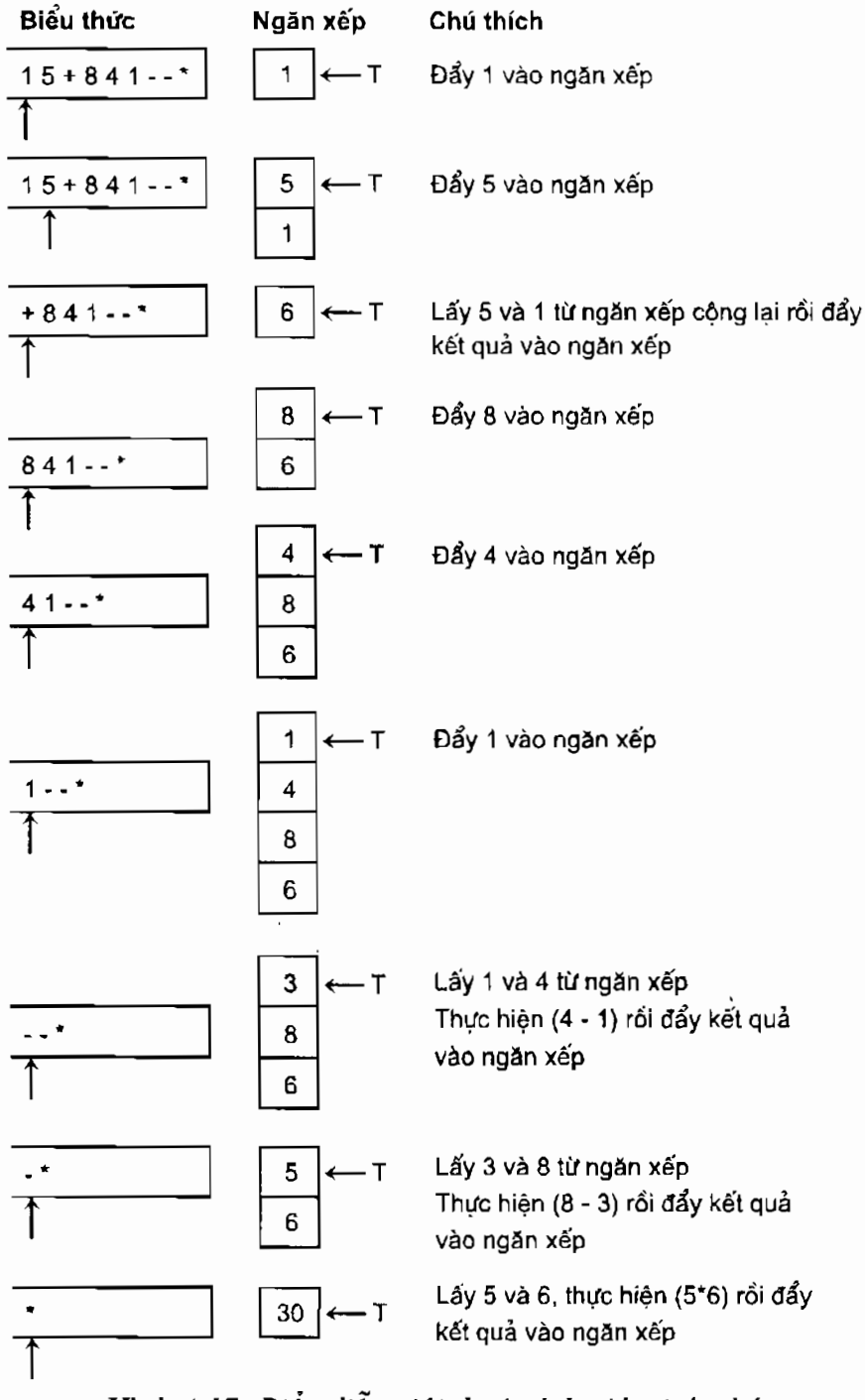
hai toán hạng cuối cùng phải được tìm ra và kết hợp với toán tử này. Như vậy ở đây đã xuất hiện cơ chế hoạt động "vào sau ra trước" nghĩa là ta sẽ phải sử dụng tới ngăn xếp để lưu trữ các toán hạng. Cứ mỗi lần đọc được một toán tử thì hai giá trị sẽ được lấy ra từ ngăn xếp để áp đặt toán tử đó lên chúng và kết quả lại được đẩy vào ngăn xếp.

Giải thuật sau đây thể hiện các ý trên.

```
void Dinh_Gia()
{
/*giải thuật này sử dụng ngăn xếp S để lưu trữ các toán hạng đọc từ biểu thức*/
  do
  {
        //Đọc phần tử X tiếp theo trong biểu thức;
        if (X là toán hạng)
              call PUSH(S, X);
              else
              {
                    call POP(S, Y);
                    call POP(S, Z);
//tác động toán tử X lên hai toán hạng Y và Z rồi gán cho biến W
              W = Y (X) Z; call PUSH(S, W);
        }
  }
  while (gặp dấu kết thúc biểu thức);
  POP(S, R);
  cout<<R;
}
```

Dưới đây là hình ảnh minh họa việc thực hiện giải thuật này với biểu thức:

1 5 + 8 4 1 - - *



Hình 4.17: Biểu diễn giải thuật tính giá trị đa thức

4.2. Queue (Hàng đợi)

4.2.1. Khái niệm

Một kiểu dữ liệu trừu tượng quan trọng khác được xây dựng trên cơ sở mô hình dữ liệu danh sách tuyến tính là hàng đợi. Hàng đợi là kiểu danh sách tuyến tính trong đó, phép bổ sung một phần tử vào hàng đợi được thực hiện ở một đầu, gọi là lối sau (rear) và phép loại bỏ một phần tử được thực hiện ở đầu kia, gọi là lối trước (front).

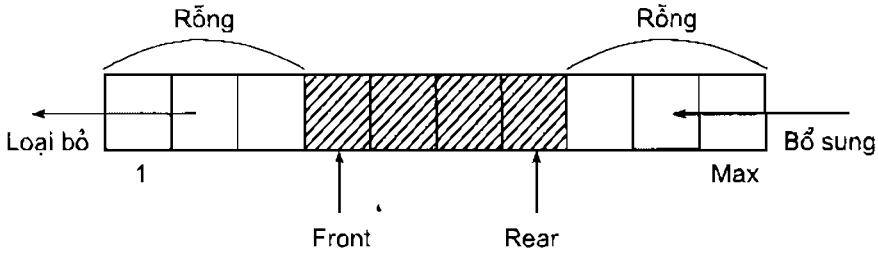
Như vậy, cơ cấu của hàng đợi là vào ở một đầu, ra ở đầu khác, phần tử vào trước thì ra trước, phần tử vào sau thì ra sau. Do đó, hàng đợi còn được gọi là danh sách kiểu ***FIFO*** (*First In First Out*). Trong thực tế ta cũng thấy có những hình ảnh giống hàng đợi, chẳng hạn, hàng người chờ mua vé tàu, học sinh xếp hàng đi vào lớp, v.v...

4.2.2. Cài đặt hàng đợi bởi mảng

Ta có thể biểu diễn hàng đợi bởi mảng với việc sử dụng hai chỉ số front để chỉ vị trí đầu hàng đợi (lối trước) và rear để chỉ vị trí cuối hàng đợi (lối sau). Cấu trúc dữ liệu hàng đợi được biểu diễn như sau:

```
#define   max N
//Khai báo kiểu dữ liệu Item nếu cần
struct Queue
{
        unsigned int front, rear;
        Item E[max];
};
struct Queue Q;
//Khai báo hàng đợi Q lưu trữ các phần tử của danh sách
```

Trong cách cài đặt như trên, hàng đợi Q là rỗng nếu Q.rear = 0 và hàng đầy nếu Q.rear = Max.



Hình 4.18: Mảng biểu diễn hàng đợi

**** Các phép toán trên hàng đợi***

Giống như ngăn xếp, khi thực hiện các thao tác với hàng đợi ta không được phép truy nhập tùy tiện vào các phần tử của hàng đợi, mà phải sử dụng các phép toán đặc biệt được định nghĩa trên hàng đợi. Đó là các phép toán sau:

a. Khởi tạo hàng đợi rỗng

```
void Initialize(struct Queue *Q)
{       Q->front = 1; Q->rear = 0;
}
```

b. Kiểm tra hàng đợi rỗng

```
int Empty(struct Queue Q)
{
        return (Q.rear ==0);
}
```

c. Kiểm tra hàng đợi đầy

```
int Full(struct Queue Q)
{
        return (Q.rear == max);
}
```

d. Bổ sung một phần tử mới vào đầu hàng đợi

Khi bổ sung một phần tử mới (với thông tin lưu trong biến X) vào hàng đợi cần kiểm tra xem hàng có đầy không, nếu hàng chưa đầy

thì bổ sung phần tử mới vào hàng, ngược lại việc bổ sung không được thực hiện.

```
int AddQ(struct Queue *Q, Item X)
{
        if (Full(*Q)) return 0;
        else
        {
                Q->rear = Q->rear + 1;
                Q->E[Q->rear] = X;
                return 1;
        }
}
```

Hàm AddQ thực hiện bổ sung phần tử mới vào cuối hàng đợi, hàm trả về 1 nếu bổ sung thành công, và trả về 0 nếu ngược lại.

e. Loại bỏ một phần tử ra khỏi hàng đợi

Khi loại bỏ một phần tử cần phải kiểm tra xem hàng đợi có rỗng không, nếu hàng đợi rỗng thì không thể thực hiện việc loại bỏ, ngược lại thì loại bỏ phần tử ở đầu hàng, nội dung của phần tử này được lưu trong biến X. Thêm nữa, khi loại bỏ, nếu hàng chỉ có một phần tử (nghĩa là hàng sẽ rỗng sau khi loại bỏ) thì cần khởi tạo lại hàng.

Sau đây là thủ tục thực hiện việc loại bỏ.

```
int DeleteQ(struct Queue *Q, Item X)
{
  if (Empty(*Q))
        return 0;
  else
  {
        X = Q->E[Q->front];
        if (Q->front == Q->rear)
        {
                Q->front = 1; Q->rear = 0;      //khởi tạo lại hàng đợi
        }
```

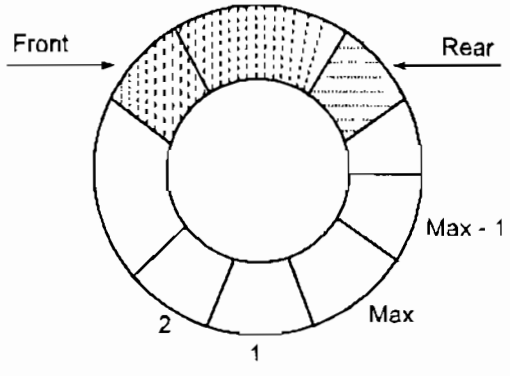
```
            else    Q->front = Q->front + 1;
            return 1;
    }
}
```

Hàm DeleteQ thực hiện lấy ra một phần tử ở đầu hàng đợi, hàm trả về 1 nếu phép lấy ra thành công, ngược lại trả về 0.

Nhận xét: Phương pháp cài đặt hàng đợi bởi mảng với hai chỉ số như trên có nhược điểm lớn. Nếu phép loại bỏ không thường xuyên làm cho hàng rỗng, thì các chỉ số front và rear sẽ tăng liên tục và sẽ vượt quá kích thước của mảng. Hàng sẽ trở thành đầy, mặc dù các vị trí trống trong mảng có thể vẫn còn nhiều (do việc loại bỏ các phần tử ở đầu hàng). Để tránh tình trạng này, mỗi khi hàng đợi đầy ta lại kiểm tra không gian nhớ phía trước hàng đợi, nếu còn trống thì đẩy hàng đợi về phía trước để tạo ra không gian nhớ trống ở phía sau. Tuy nhiên việc này tiêu tốn rất nhiều thời gian.

4.2.3. Cài đặt hàng đợi bởi mảng vòng tròn

Để khắc phục nhược điểm nêu trên, người ta đưa ra phương pháp cài đặt hàng đợi bởi mảng vòng tròn. Đó là một mảng với chỉ số chạy trong miền 1..max, với mọi i = 1, 2,..., max – 1, phần tử thứ i của mảng đi trước phần tử thứ i + 1, còn phần tử thứ max đi trước phần tử đầu tiên, tức là các phần tử của mảng được xếp thành vòng tròn (xem hình 4.20).



Hình 4.20: Hàng đợi vòng tròn

Khi biểu diễn hàng bởi mảng vòng tròn, để biết khi nào hàng đầy, khi nào hàng rỗng ta cần đưa thêm vào biến count để đếm số phần tử trong hàng. Chúng ta có khai báo cấu trúc dữ liệu sau:

```
#define   max N
//Khai báo kiểu dữ liệu Item (nếu cần)
struct    Queue
{
        int count;
        int front, rear;
        Item E[max];
};
struct Queue Q;
```

Với cách cài đặt này, hàng rỗng khi Q.count = 0, hàng đầy khi Q.count = max.

Khi làm việc với mảng vòng tròn, cần lưu ý rằng, phần tử đầu tiên của mảng đi sau phần tử thứ max. Sau đây chúng ta sẽ cài đặt thao tác bổ sung một phần tử vào hàng đợi, và loại bỏ một phần tử khỏi hàng đợi, các thao tác khác dành cho bạn đọc.

```
void AddQ(struct Queue *Q, Item X)
{
        if (Q->count == max) cout<<"\nHang day"<<endl;
        else
        {
                if (Q->rear == max)
                        Q->rear = 1;
                else
                        Q->rear = Q->rear + 1;
                        Q->E[Q->rear] = X;
                        Q->count = Q->count + 1;
        }
}
```

```
void DeleteQ(struct Queue *Q, Item X)
{
        if (Q->count = = 0) cout<<"\nhang rong"<<endl;
        else {
                X = Q->E[Q->front];
                if (Q->front = = Q->rear)
                {       Q->front = 1; Q->rear = 0;
                }
                else
                        if (Q->front = = max) Q->front = 1;
                        else Q->front = Q->front + 1;
                Q->count = Q->count – 1;
        }
}
```

Hàng đợi thường dùng để thực hiện các "tuyến chờ" (waiting lines) trong xử lý động, đặc biệt trong các hệ mô phỏng (simulation), đó là các hệ mô hình hoá các quá trình động và người ta dùng mô hình này để nghiên cứu hoạt động của các quá trình ấy.

4.2.4. Cài đặt hàng đợi bởi danh sách móc nối đơn

Như ta đã biết, đối với ngăn xếp việc truy nhập chỉ được thực hiện ở một đầu (đỉnh). Vì vậy, việc cài đặt ngăn xếp bằng danh sách móc nối là khá tự nhiên. Chẳng hạn, với danh sách móc nối đơn trỏ bởi L thì có thể coi L như con trỏ trỏ tới đỉnh của ngăn xếp. Bổ sung một nút vào ngăn xếp chính là việc bổ sung một nút vào thành nút đầu tiên của danh sách, còn loại bỏ một nút ra khỏi ngăn xếp chính là loại bỏ nút đầu tiên của danh sách đang trỏ bởi L. Trong việc bổ sung với ngăn xếp dạng này không cần kiểm tra hiện tượng tràn như với ngăn xếp lưu trữ kế tiếp.

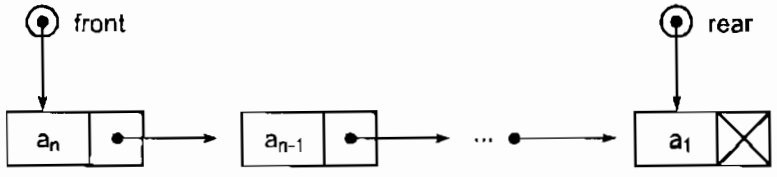
Đối với hàng đợi thì loại bỏ ở một đầu, còn bổ sung thì ở đầu kia. Nếu coi danh sách móc nối đơn như một hàng đợi thì việc loại bỏ một nút cũng giống như với ngăn xếp, nhưng bổ sung một nút thì phải thực gắn nút mới vào cuối hàng đợi, nghĩa là phải tìm đến nút cuối

cùng. Trong trường hợp này, để lưu trữ danh sách người ta dùng hai con trỏ, một con trỏ trỏ vào nút đầu danh sách và một con trỏ trỏ vào nút cuối danh sách.

Cấu trúc dữ liệu biểu diễn hàng đợi như sau:

```
//Khai báo kiểu dữ liệu Item (nếu cần)
struct   Node
{        Item Infor;
         struct Node *Next;
};
struct   Queue
{
         struct Node *front;
         struct Node *rear;
};
struct Queue Q;
```

Trong cách biểu diễn trên, front là con trỏ trỏ đến nút đầu hàng đợi, rear là con trỏ trỏ đến nút cuối hàng đợi.



Hình 4.21: Danh sách móc nối biểu diễn hàng đợi

Với cách cài đặt này, hàng đợi được xem là không khi nào đầy. Hàng đợi rỗng khi Q.front = NULL.

Sau đây là các hàm và thủ tục thực hiện các phép toán trên hàng đợi.

a. Khởi tạo hàng đợi rỗng:

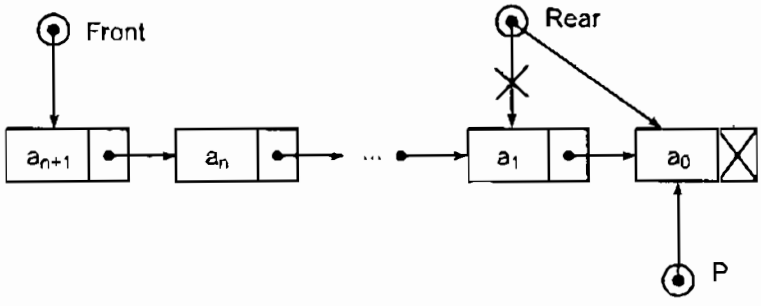
```
void Create(struct Queue *Q)
{
         Q->front = NULL;
         Q->rear = NULL;
}
```

b. Kiểm tra hàng đợi rỗng:

```
int Empty(struct Queue Q)
{
        return (Q.front = = NULL);
}
```

c. Bổ sung một phần tử vào cuối hàng đợi:

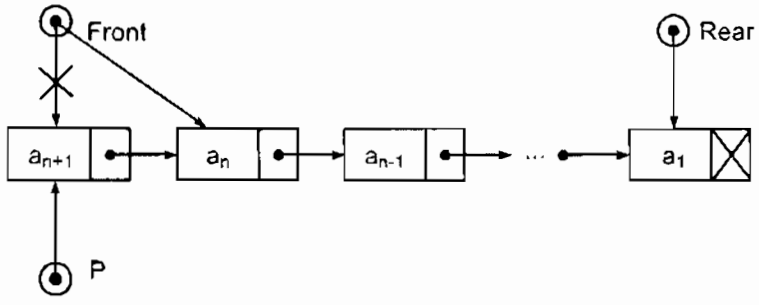
```
void ADD(struct Queue *Q, Item X)
{
  struct Node *P;
        P = (struct Node*)malloc(sizeof(struct Node));
        P->Infor = X;
        P->Next = NULL;
        if (Empty(*Q))
        {       Q->front = P;
                Q->rear = P;
        }
        else    {
                Q->rear->Next = P;
                Q->rear = P;
                }
}
```



Hình 4.22: Minh họa thao tác bổ sung

d. Lấy ra một phần tử ở đầu hàng đợi

```
int Del(struct Queue *Q, Item X)
{
  struct Node P;
  if (Empty(*Q)) return 0;
  else    {
                P = Q->front;
                X = Q->front->Infor;
                Q->front = Q->front->Next;
                free(P);
                return 1;
          }
}
```

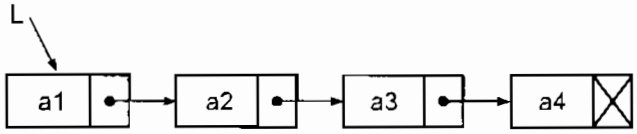


Hình 4.23: Minh họa thao tác lấy ra một phần tử

Trên đây chúng ta đã xem xét một loại cấu trúc dữ liệu, được sử dụng rất phổ biến trong các ứng dụng, đó là danh sách tuyến tính với các dạng khác nhau được cài đặt theo hai cách: bằng mảng (lưu trữ kế tiếp) và bằng con trỏ (lưu trữ móc nối), cùng với các phép toán xử lý tương ứng trên mỗi loại. Bạn đọc cũng được tìm hiểu hai cách thức lưu trữ đặc biệt đó là danh sách kiểu ngăn xếp và kiểu hàng đợi. Mỗi loại danh sách và cách cài đặt chúng có những ưu và nhược điểm khac nhau. Tuy nhiên, để hiểu rõ hơn về nó, ta cần cài đặt một số ứng dụng nhỏ trên máy đối với mỗi loại. Khi cài đặt hãy lựa chọn cấu trúc lưu trữ phù hợp với dữ liệu của bài toán.

BÀI TẬP CHƯƠNG 4

Bài 1: Cho danh sách nối đơn với nút đầu tiên được trỏ bởi con trỏ L như hình vẽ:



Yêu cầu:

a. Vẽ hình mô tả trạng thái của danh sách trong quá trình tạo mới danh sách gồm 4 phần tử như trên, xuất phát từ một danh sách rỗng (yêu cầu mô tả từng bước trong quá trình).

b. Vẽ hình mô thao tác loại bỏ phần tử đầu tiên (a1) và phần tử thứ 3 (a3) trong danh sách (cần chú thích rõ ràng).

c. Vẽ hình mô tả thao tác bổ sung phần tử 'a5' vào đầu danh sách và vào sau phần tử thứ 3 (a3) trong danh sách.

d. Giả sử a3 là nhỏ nhất, vẽ hình mô tả tình trạng của danh sách trong thao tác sắp xếp tăng dần bằng phương pháp lựa chọn ở lần duyệt đầu tiên (chỉ vẽ trạng thái lúc bắt đầu duyệt, lúc chuẩn bị đổi chỗ và sau khi đổi chỗ).

e. Giả sử danh sách lưu trữ thông tin về các sinh viên, mỗi sinh viên gồm: Mã sinh viên, họ và tên, năm sinh, điểm tổng kết. Cài đặt chương trình thực hiện các yêu cầu sau:

- Khai báo cấu trúc dữ liệu của danh sách.
- Tạo mới danh sách gồm 10 phần tử (các thông tin được nhập từ bàn phím). Hoặc tạo mới danh sách với các thông tin phần tử nhập từ bàn phím, việc nhập kết thúc khi tên của sinh viên nhập vào là xâu rỗng.
- Hiển thị danh sách lên màn hình.
- Xóa phần tử đầu tiên trong danh sách, hiển thị lại danh sách
- Xác định tỷ lệ sinh viên giỏi (dtk >= 8), khá (dtk >= 6.5), trung bình (dtk >= 5), yếu (còn lại).

- Xóa phần tử thứ 4 trong danh sách, hiển thị lại danh sách.
- Thêm một phần tử vào đầu danh sách, hiển thị lại danh sách
- Thêm một phần tử vào sau phần tử thứ 3 trong danh sách, hiển thị lại danh sách.
- Tìm sinh viên có tên "Doanh" trong danh sách, hiển thị kết quả tìm kiếm (nếu có hiển thị thông tin đầy đủ của sinh viên này).
- Sắp xếp danh sách theo chiều giảm dần của điểm tổng kết, hiển thị lại danh sách.
- Thêm thông tin một học sinh mới vào danh sách sao cho trật tự vừa sắp không bị thay đổi.

Bài 2: Thực hiện lại các yêu cầu của bài 1 với danh sách nối đôi.

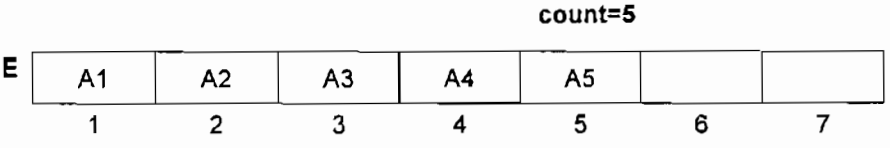
Bài 3: Cài đặt chương trình thực hiện các yêu cầu:

- Tạo một danh sách nối đơn/nối đôi lưu trữ các số nguyên, dữ liệu được nhập từ bàn phím, việc nhập kết thúc khi số nguyên nhập vào là -1 (lưu ý -1 không phải là một phần tử của danh sách) hoặc cho phép nhập n số với n nhập từ bàn phím.
- Hiển thị danh sách lên màn hình.
- Bổ sung số nguyên X vào vị trí K trong danh sách (X, K nhập từ bàn phím), hiển thị lại danh sách.
- Xóa số thứ k trong danh sách (k nhập từ bàn phím), hiển thị lại danh sách.
- Xóa các số âm trong danh sách, hiển thị lại danh sách.
- Sắp xếp danh sách theo chiều tăng dần/giảm dần, hiển thị lại danh sách.
- Nhập một số nguyên, bổ sung nó vào danh sách sao cho danh sách vẫn có thứ tự, hiển thị lại danh sách.
- Cho biết chiều dài của danh sách, số lượng số âm, số lượng số dương.

Bài 4: Cài đặt chương trình thực hiện các yêu cầu sau:

- Tạo một danh sách nối đơn/nối đôi lưu trữ các phân số có tử số và mẫu số là các số nguyên khác không, dữ liệu được nhập từ bàn phím, việc nhập kết thúc nếu gặp phân số mà tử số hoặc mẫu số của nó được nhập là số không.
- Hiển thị danh sách lên màn hình (ví dụ: 3/4, -2/5, 1/2,...)
- Cho biết trong danh sách trên có bao nhiêu phân số chưa được tối giản, hãy tối giản các phân số đó, hiển thị lại danh sách.
- Bổ sung một phân số mới vào vị trí thứ k trong danh sách (phân số mới và k được nhập từ bàn phím), hiển thị lại danh sách.
- Tính tổng các phân số có cùng mẫu số là 9 trong danh sách, hiển thị kết quả sau khi đã tối giản.
- Xóa các phân số có tử số là số âm, hiển thị lại danh sách.

Bài 5: Danh sách L được lưu trữ kế tiếp biểu diễn bởi hình vẽ (E: mảng lưu các phần tử của danh sách, count biến nguyên lưu độ dài thực của danh sách.



Yêu cầu:

a. Vẽ hình mô tả trạng thái của danh sách qua các bước trong quá trình tạo mới danh sách từ một danh sách rỗng.

b. Vẽ hình mô tả trạng thái của danh sách trong thao tác loại bỏ phần tử đầu tiên (A1) và phần tử thứ 3 (A3) trong danh sách (cần chú thích rõ ràng).

c. Vẽ hình mô tả quá trình bổ sung phần tử A6 vào đầu danh sách, vào sau phần tử thứ 3 (A3) trong danh sách.

d. Giả sử danh sách lưu trữ thông tin về các sinh viên, mỗi sinh viên gồm: Mã sinh viên, họ và tên, năm sinh, điểm tổng kết. Hãy cài đặt chương trình thực hiện các yêu cầu sau:
   - Khai báo cấu trúc dữ liệu của danh sách
   - Nhập mới 10 phần tử cho danh sách
   - Hiển thị danh sách lên màn hình
   - Xóa phần tử đầu tiên trong danh sách, hiển thị lại danh sách
   - Xóa phần tử thứ 4 trong danh sách, hiển thị lại danh sách
   - Thêm một phần tử vào đầu danh sách, hiển thị lại danh sách
   - Thêm một phần tử vào sau phần tử thứ 3 trong danh sách, hiển thị danh sách
   - Tìm sinh viên có tên "Doanh" trong danh sách, hiển thị kết quả tìm kiếm.
   - Sắp xếp danh sách theo chiều giảm dần của điểm tổng kết, hiển thị lại danh sách.

Bài 6: Cài đặt chương trình thực hiện các yêu cầu:

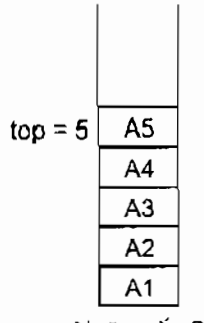
- Tạo một danh sách lưu trữ kế tiếp các số nguyên, dữ liệu được nhập từ bàn phím, việc nhập kết thúc khi số nguyên nhập vào là -1.
- Hiển thị danh sách lên màn hình.
- Bổ sung số nguyên X vào vị trí K trong danh sách (X, K nhập từ bàn phím), hiển thị lại danh sách.
- Xóa số thứ I trong danh sách (I nhập từ bàn phím), hiển thị lại danh sách.
- Xóa các số âm trong danh sách, hiển thị lại danh sách.
- Sắp xếp danh sách theo chiều tăng dần/giảm dần, hiển thị lại danh sách.
- Nhập một số nguyên, bổ sung nó vào danh sách sao cho danh sách vẫn có thứ tự, hiển thị lại danh sách.

Bài 7: Cài đặt chương trình thực hiện các yêu cầu sau:

- Tạo một danh sách lưu trữ kế tiếp các phân số có tử số và mẫu số là các số nguyên khác không, dữ liệu được nhập từ bàn phím, việc nhập kết thúc nếu gặp phân số mà tử số hoặc mẫu số của nó được nhập là số không.
- Hiển thị danh sách lên màn hình (ví dụ: 3/4, -2/5, 1/2,...).
- Cho biết trong danh sách trên có bao nhiêu phân số chưa được tối giản, hãy tối giản các phân số đó, hiển thị lại danh sách.
- Bổ sung một phân số mới vào vị trí thứ k trong danh sách (phân số mới và k được nhập từ bàn phím), hiển thị lại danh sách.
- Tính tổng các phân số có cùng mẫu số là 9 trong danh sách, hiển thị kết quả sau khi đã tối giản.
- Xóa các phân số có tử số là số âm, hiển thị lại danh sách

Bài 8: Cho ngăn xếp S được biểu diễn bởi hình vẽ.

- Vẽ hình mô tả trạng thái của ngăn xếp qua các bước, khi thực hiện thao tác loại bỏ phần tử A2 trong ngăn xếp.
- Vẽ hình mô tả trạng thái của ngăn xếp qua các bước khi thực hiện thao tác bổ sung phần tử A6 vào đáy ngăn xếp.
- Giả sử ngăn xếp được lưu trữ bởi một danh sách nối đơn/lưu trữ kế tiếp với mỗi phần tử là một số nguyên.



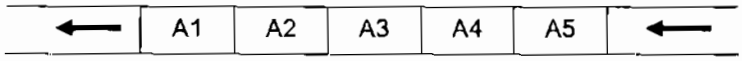
Ngăn xếp S

Yêu cầu:

- Cài đặt cấu trúc dữ liệu của ngăn xếp cùng với các phép toán cơ sở của nó.
- Cài đặt phép toán loại bỏ phần tử thứ 2 trong ngăn xếp tính từ đáy.

- Cài đặt phép toán bổ sung 1 phần tử vào đáy ngăn xếp
- Cài đặt phép toán loại bỏ phần tử thứ k trong ngăn xếp.

Bài 9: Cho hàng đợi Q được biểu diễn bởi hình vẽ sau:



- Vẽ hình mô tả trạng thái của hàng đợi khi thực hiện thao tác bổ sung phần tử A6 vào đầu hàng đợi, và bổ sung phần tử A7 vào sau phần tử A3
- Vẽ hình mô tả trạng thái của hàng đợi khi thực hiện thao tác loại bỏ phần tử A3 và phần tử A5 trong hàng đợi.
- Giả sử hàng đợi được lưu trữ kế tiếp/lưu trữ bởi một danh sách nối đơn, mỗi phần tử chứa một số nguyên.

Yêu cầu:

- Cài đặt cấu trúc dữ liệu của hàng đợi cùng với các phép toán cơ sở của nó
- Cài đặt phép toán bổ sung một phần tử vào đầu hàng
- Cài đặt phép toán bổ sung một phần tử vào sau phần tử thứ 3 tính từ đầu hàng
- Cài đặt phép toán loại bỏ phần tử thứ k trong hàng.

Chương 5

CÂY

Trong chương này chúng ta sẽ nghiên cứu mô hình dữ liệu cây (tree). Cây là một cấu trúc phân cấp trên một tập hợp nào đó các đối tượng. Một ví dụ quen thuộc về cây, đó là cây thư mục hoặc mục lục của cuốn sách cũng là một cây. Cây được sử dụng rộng rãi trong rất nhiều vấn đề khác nhau. Chẳng hạn nó được áp dụng để tổ chức thông tin trong các hệ cơ sở dữ liệu, để mô tả cấu trúc cú pháp của các chương trình nguồn khi xây dựng các chương trình dịch. Rất nhiều bài toán mà ta gặp trong các lĩnh vực khác nhau được quy về việc thực hiện các phép toán trên cây. Trong chương 4 chúng ta sẽ trình bày định nghĩa, các khái niệm cơ bản về cây. Chúng ta cũng sẽ xét các phương pháp cài đặt cây và thực hiện các phép toán cơ bản trên cây. Sau đó ta nghiên cứu kỹ một số dạng cây đặc biệt đó là cây nhị phân tìm kiếm và cây cân bằng.

1. CÂY VÀ CÁC KHÁI NIỆM LIÊN QUAN

1.1. Định nghĩa

Định nghĩa 1: Một cây là tập hợp hữu hạn các nút trong đó có một nút đặc biệt gọi là gốc (root). Giữa các nút có mối quan hệ phân cấp gọi là quan hệ "cha-con".

Định nghĩa 2: Cây được định nghĩa đệ quy như sau:

- Một nút là một cây và nút này cũng là gốc của cây.

- Giả sử T_1, T_2,..., T_n ($n \geq 1$) là các cây có gốc tương ứng r_1, r_2,..., r_n. Khi đó cây T với gốc r được hình thành bằng cách cho r trở thành nút cha của các nút r_1, r_2,..., r_n

1.2. Một số khái niệm cơ bản

- *Bậc của một nút*: là số con của nút đó

- *Bậc của một cây*: là bậc của nút có bậc lớn nhất trên cây đó (số cây con tối đa của một nút thuộc cây). Cây có bậc n thì gọi là cây n - phân

- *Nút gốc*: là nút có không có nút cha

- *Nút lá*: là nút có bậc bằng 0

- *Nút nhánh*: là nút có bậc khác 0 và không phải là nút gốc

- *Mức của một nút*

Mức (gốc (T_0)) =1

Gọi T_1, T_2,..., T_n là các cây con của T_0.

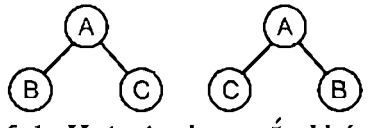
Khi đó Mức (T1) = Mức (T2) = ... = Mức (Tn) = Mức (T0) +1

- *Chiều cao của cây*: là mức của nút có mức lớn nhất có trên cây đó

- *Đường đi*: Dãy các đỉnh n_1, n_2,..., n_k được gọi là đường đi nếu n_i là cha của n_{i+1} ($1 \le i \le k-1$)

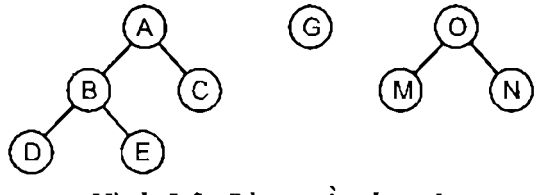
- *Độ dài của đường đi*: là số nút trên đường đi -1

- *Cây được sắp thứ tự*: Trong một cây, nếu các cây con của mỗi đỉnh được sắp theo một thứ tự nhất định, thì cây được gọi là cây được sắp (cây có thứ tự). Chẳng hạn, hình 5.1 minh họa hai cây được sắp khác nhau.



Hình 5.1: Hai cây được sắp khác nhau

- *Rừng*: là tập hợp hữu hạn các cây phân biệt.



Hình 5.2: Rừng gồm ba cây

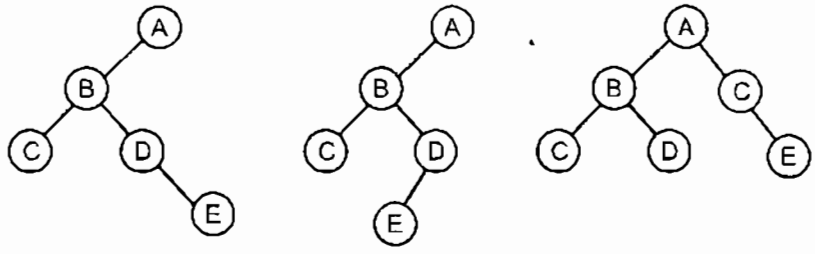
Sau đây ta sẽ tìm hiểu một loại cây đặc biệt được gọi là cây nhị phân.

2. CÂY NHỊ PHÂN

2.1. Định nghĩa

Cây nhị phân là cây mà mỗi nút có tối đa hai cây con. Đối với cây con của một nút người ta cũng phân biệt cây con trái và cây con phải.

Như vậy cây nhị phân là cây có thứ tự.



Hình 5.3: Một số dạng cây nhị phân

2.2. Tính chất

Đối với cây nhị phân cần chú ý tới một số tính chất sau:

1. Số lượng tối đa các nút có ở mức i trên cây nhị phân là 2^{i-1} ($i \geq 1$)

2. Số lượng nút tối đa trên một cây nhị phân có chiều cao h là 2^h-1 ($h \geq 1$).

**** Chứng minh***

+ Tính chất 1 sẽ được chứng minh bằng qui nạp

Bước cơ sở: với i = 1, cây nhị phân có $1 = 2^0$ nút. Vậy mệnh đề đúng với i = 1.

Bước qui nạp: Giả sử kết quả đúng với mức i, nghĩa là ở mức này cây nhị phân có tối đa 2^{i-1} nút, ta chứng minh mệnh đề đúng với mức i + 1.

Theo định nghĩa cây nhị phân thì tại mỗi nút có tối đa hai cây con nên mỗi nút ở mức i có tối đa hai con. Do đó theo giả thiết qui nạp ta suy ra tại mức i + 1 ta có:

$$2^{i-1} \times 2 = 2^i \text{ nút.}$$

+ Tính chất 2 được chứng minh như sau:

Ta đã biết rằng chiều cao của cây là số mức lớn nhất có trên cây đó. Theo tính chất 1 ta suy ra số nút tối đa có trên cây nhị phân với chiều cao h là:

$$2^0 + 2^1 + ... + 2^{h-1} = 2^h - 1.$$

Từ kết quả này có thể suy ra:

Nếu cây nhị phân có n nút thì chiều cao của nó là $h = \lceil \log_2(n+1) \rceil$

(Ta qui ước: $\lceil x \rceil$ là số nguyên trên của x

$\lfloor x \rfloor$ là số nguyên dưới của x).

2.3. Biểu diễn cây nhị phân

2.3.1. Lưu trữ kế tiếp

Phương pháp tự nhiên nhất để biểu diễn cây nhị phân là chỉ ra đỉnh con trái và đỉnh con phải của mỗi đỉnh.

Ta có thể sử dụng một mảng để lưu trữ các đỉnh của cây nhị phân. Mỗi đỉnh của cây được biểu diễn bởi bản ghi gồm ba trường:

Infor: mô tả thông tin gắn với mỗi đỉnh

Left: chỉ đỉnh con trái

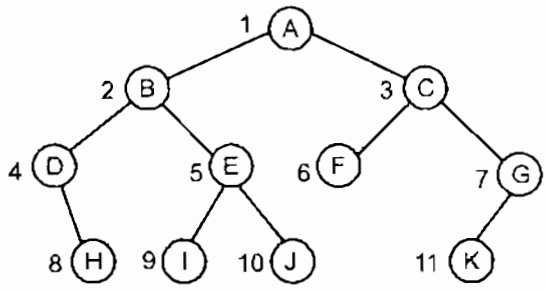
Right: chỉ đỉnh con phải.

Giả sử các đỉnh của cây được được đánh số từ 1 đến max, dữ liệu của các đỉnh trên cây có kiểu là ***Item***. Khi đó cấu trúc dữ liệu biểu diễn cây nhị phân được khai báo như sau:

```
#define max = N;          //Số thứ tự lớn nhất của nút trên cây
//Khai báo kiểu dữ liệu Item (nếu cần)
struct Node
{
        Item Infor;
        int Letf;
```

```
        int Right;
};
struct Node T[max];      //T là mảng lưu trữ các nút của cây.
```

Ví dụ:



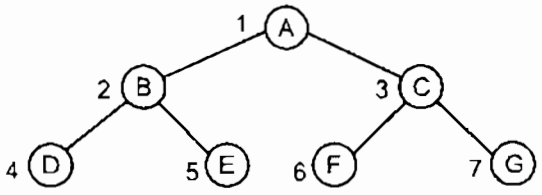
Hình 5.4: Một cây nhị phân

Hình 5.5 minh hoạ cấu trúc dữ liệu biểu diễn cây nhị phân trong hình 5.4.

	infor	Left	Right
1	A	2	3
2	B	4	5
3	C	6	7
4	D	0	8
5	E	9	10
6	F	0	0
7	G	11	9
8	H	0	0
9	I	0	0
10	J	0	0
11	K	0	0

Hình 5.5: Cấu trúc dữ liệu biểu diễn cây

Nếu có một cây nhị phân hoàn chỉnh đầy đủ, ta có thể dễ dàng đánh số cho các nút trên cây đó theo thứ tự lần lượt từ mức 1 trở lên, hết mức này đến mức khác và từ trái qua phải đối với các nút ở mỗi mức. Ví dụ hình 5.6 minh họa cây nhị phân được đánh số.



Hình 5.6: Cây nhị phân được đánh số

Ta có nhận xét sau: con của nút thứ i là các nút thứ 2i và 2i + 1 hoặc cha của nút thứ j là $\lfloor j/2 \rfloor$.

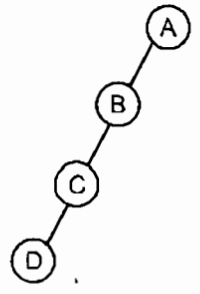
Như vậy, ta có thể lưu trữ cây này bằng một vectơ V, theo nguyên tắc: nút thứ i của cây được lưu trữ ở V[i]. Đó chính là cách lưu trữ kế tiếp đối với cây nhị phân. Với cách lưu trữ này nếu biết được địa chỉ của nút con sẽ tính được địa chỉ nút cha và ngược lại.

Với cây đầy đủ nêu trên thì hình ảnh lưu trữ sẽ như sau:

A	B	C	D	E	F	G
v[1]	v[2]	v[3]	v[4]	v[5]	v[6]	v[7]

Nhận xét:

Nếu cây nhị phân không đầy đủ thì cách lưu trữ này không thích hợp vì sẽ gây ra lãng phí bộ nhớ do có nhiều phần tử bỏ trống (ứng với cây con rỗng). Ta hãy xét cây như hình 5.6. Để lưu trữ cây này ta phải dùng mảng gồm 15 phần tử mà chỉ có 5 phần tử khác rỗng, hình ảnh lưu trữ miền nhớ của cây này (hình 5.7).



Hình 5.7: Cây nhị phân đặc biệt

	B	∅	C	∅	∅	∅	D	∅	∅	∅	∅	∅	∅	∅	E	∅	...

(∅: chỉ chỗ trống)

Nếu cây nhị phân luôn biến động nghĩa là có phép bổ sung, loại bỏ các nút thường xuyên tác động thì cách lưu trữ này gặp phải một số nhược điểm như tốn thời gian khi phải thực hiện các thao tác này, độ cao của cây phụ thuộc vào kích thước của mảng,...

2.3.2. Lưu trữ móc nối

Cách lưu trữ móc nối khắc phục được những nhược điểm của cách lưu trữ kế tiếp đồng thời phản ánh được dạng tự nhiên của cây.

Trong cách lưu trữ móc nối, mỗi nút tương ứng với một phần tử nhớ có qui cách như sau:

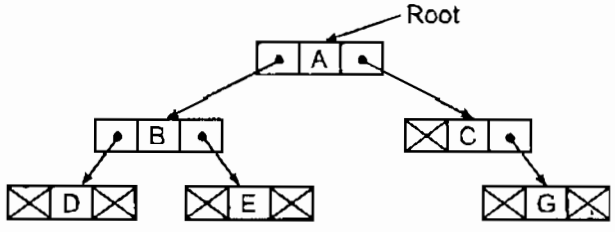
Letf	Infor	Right

- Trường Infor ứng với thông tin (dữ liệu) của nút
- Trường Left ứng với con trỏ, trỏ tới cây con trái của nút đó
- Trường Right ứng với con trỏ, trỏ tới cây con phải của nút đó

Ta có thể khai báo cấu trúc dữ liệu như sau:

```
struct Node
 {      Item infor;
        struct Node *Left, *Right;
 };
struct Node *Root; //Con trỏ Root trỏ vào nút gốc cây
```

Ví dụ: cây nhị phân hình 5.6 có dạng lưu trữ móc nối như hình 5.8.



Hình 5.8: Cấu trúc dữ liệu biểu diễn cây

Để lưu trữ và thao tác với cây, cần một con trỏ Root, trỏ tới nút gốc của cây.

2.4. Phép duyệt cây nhị phân

Phép xử lý các nút trên cây - mà ta gọi chung là phép "thăm" các nút một cách hệ thống, sao cho mỗi nút được thăm đúng một lần theo một thứ tự xác định, gọi là phép duyệt cây. Có thể duyệt cây nhị phân

theo một trong ba thứ tự: duyệt trước, duyệt giữa và duyệt sau, các phép duyệt này được định nghĩa đệ quy như sau:

2.4.1. Duyệt theo thứ tự trước (preorder traversal)

Nếu cây khác rỗng

- Thăm gốc
- Duyệt cây con trái theo thứ tự trước
- Duyệt cây con phải theo thứ tự trước.

2.4.2. Duyệt theo thứ tự giữa (inorder traversal)

Nếu cây khác rỗng

- Duyệt cây con trái theo thứ tự giữa
- Thăm gốc
- Duyệt cây con phải theo thư tự giữa.

2.4.3. Duyệt theo thứ tự sau (postorder traversal)

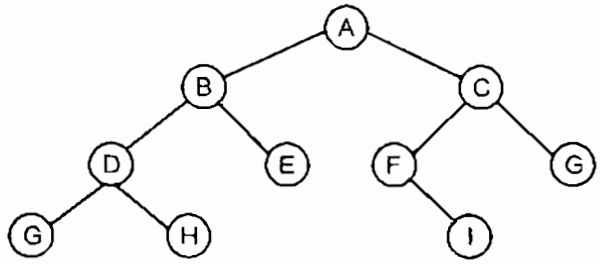
Nếu cây khác rỗng

- Duyệt cây con trái theo thứ tự sau
- Duyệt cây con phải theo thư tự sau
- Thăm gốc.

Tương ứng với ba phép duyệt ta có ba thủ tục duyệt cây nhị phân. Sau đây là thủ tục đệ quy duyệt cây theo thứ tự trước:

```
void PreOrder (struct Node *Root)
{
        if (Root != NULL)
        {
                visit(Root);
                PreOrder(Root->Left);
                PreOrder(Root->Right);
        }
}
```

Một cách tương tự, ta có thể viết được các thủ tục đệ quy đi qua cây theo thứ tự giữa và theo thứ tự sau.



Hình 5.9: Duyệt cây nhị phân

Với cây nhị phân hình 5.9, dãy các nút được thăm trong các phép duyệt là:

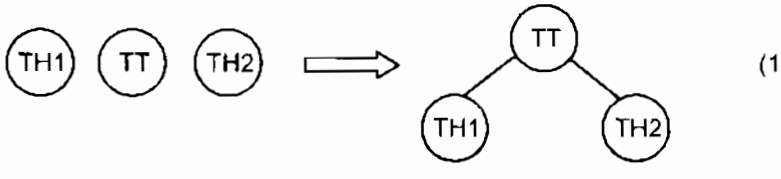
- *Duyệt theo thứ tự trước:* A B D G H E C F I G
- *Duyệt theo thứ giữa:* G D H B E A F I C G
- *Duyệt theo thứ tự sau:* G H D E B I F G C A

2.5. Cây nhị phân biểu diễn biểu thức

Cây biểu thức là cây nhị phân mà nút gốc và các nút nhánh chứa các toán tử (phép toán) còn các nút lá thì chứa các toán hạng.

2.5.1. Cách dựng cây biểu thức

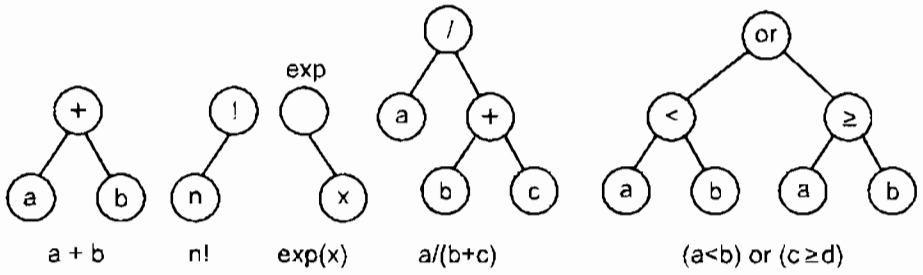
Đối với phép toán hai ngôi (chẳng hạn +, -, *, /) được biểu diễn bởi cây nhị phân mà gốc của nó chứa toán tử, cây con trái biểu diễn toán hạng bên trái, còn cây con bên phải biểu diễn toán hạng bên phải.



Hình 5.10: Biểu diễn phép toán hai ngôi

Đối với phép toán một ngôi như "phủ định" hoặc "phép toán đổi dấu", hoặc các hàm chuẩn như exp() hoặc cos()... thì cây con bên trái

rỗng. Còn với các phép toán một toán hạng như phép "lấy đạo hàm" ()' hoặc hàm "giai thừa" ()! thì cây con bên phải rỗng.



Hình 5.11: Một số cây biểu thức

Nhận xét:

Nếu ta đuyệt cây biểu thức theo thứ tự trước thì ta được biểu thức Ba Lan dạng tiền tố (pre-fix). Nếu duyệt cây nhị phân theo thứ tự sau thì ta có biểu thức Ba Lan dạng hậu tố (post-fix); còn theo thứ giữa thì ta nhận được cách viết thông thường của biểu thức (dạng trung tố).

2.5.2. Ví dụ

Để minh cho nhận xét này ta lấy ví dụ sau:

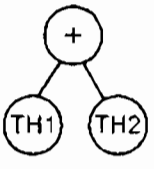
Cho biểu thức P = a*(b - c) + d/e

1. Hãy dựng cây biểu thức biểu diễn biểu thức trên

2. Đưa ra biểu thức ở dạng tiền tố và hậu tố

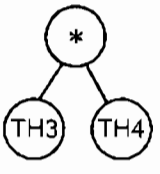
Giải:

1. Ta có $\left.\begin{matrix} TH1 = a*(b - c) \\ TH1 = d/e \end{matrix}\right\}$ suy ra cây biểu thức có dạng:

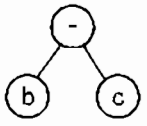


Xét TH1 = a*(b - c), toán hạng chưa ở dạng cơ bản ta phải phân tích để được như ở (1)

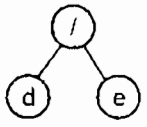
$\left.\begin{matrix} TH3 = a \\ TH4 = b - c \end{matrix}\right\}$ cây biểu thức của toán hạng này là:



Toán hạng TH4 đã ở dạng cơ bản nên ta có ngay cây biểu thức:



Cũng tương tự như vậy đối với toán hạng TH2, cây biểu thức tương ứng với toán hạng này là:



Tổng hợp cây biểu thức của các toán hạng ta được cây biểu thức (hình 5.12).

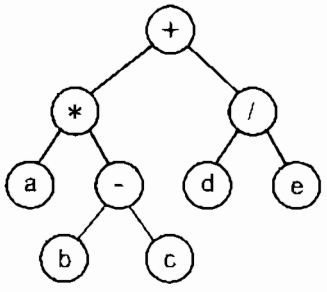
Bây giờ ta duyệt cây biểu thức ở hình 5.12.

Duyệt theo thứ tự trước:

+ * a - b c / d e

Duyệt theo thứ sau:

a b c - * d e / +



Hình 5.12: Cây biểu thức

3. CÂY TỔNG QUÁT

3.1. Biểu diễn cây tổng quát

Đối với cây tổng quát cấp m nào đó có thể sử dụng cách biểu diễn móc nối tương tự như đối với cây nhị phân. Như vậy ứng với mỗi nút ta phải dành ra m trường mối nối để trỏ tới các con của nút đó và như vậy số mối nối không sẽ rất nhiều: nếu cây có n nút sẽ có tới n(m-1) + 1 "mối nối không" trong số m×n mối nối.

Nếu tuỳ theo số con của từng nút mà định ra mối nối, nghĩa là dùng nút có kích thước biến đổi thì sự tiết kiệm không gian nhớ này sẽ phải trả giá bằng những phức tạp của quá trình xử lý trên cây.

Để khắc phục các nhược điểm trên, ta dùng cách biểu diễn cây nhị phân để biểu diễn cây tổng quát.

Ta có thể biến đổi một cây bất kỳ thành một cây nhị phân theo qui tắc sau:

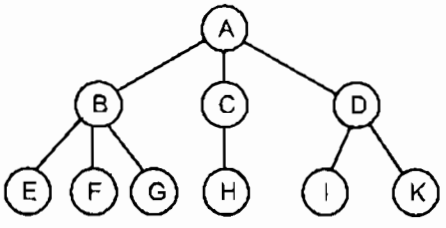
- Giữ lại nút con trái nhất làm nút con trái

- Các nút con còn lại chuyển thành các nút con phải

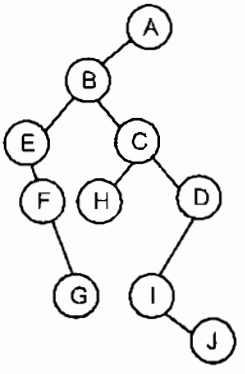
- Như vậy, trong cây nhị phân mới, con trái thể hiện quan hệ cha con và con phải thể hiện quan hệ anh em trong cây tổng quát ban đầu. Khi đó cây nhị phân này được gọi là cây nhị phân tương đương.

Ta có thể xem ví dụ dưới đây để thấy rõ qui trình. Giả sử có cây tổng quát như hình 5.13.

Cây nhị phân tương đương sẽ như hình 5.14.



Hình 5.13: Cây tổng quát



Hình 5.14: Cây nhị phân tương đương

3.2. Phép duyệt cây tổng quát

Phép duyệt cây tổng quát cũng được đặt ra tương tự như đối với cây nhị phân. Tuy nhiên, có một điều cần phải xem xét thêm, khi định nghĩa phép duyệt, đó là:

1. Sự nhất quán về thứ tự các nút được thăm giữa phép duyệt cây tổng quát và phép duyệt cây nhị phân tương đương của nó.

2. Sự nhất quán giữa định nghĩa phép duyệt cây tổng quát với định nghĩa phép duyệt cây nhị phân. Vì cây nhị phân cũng có thể coi là cây tổng quát và ta có thể áp dụng định nghĩa phép duyệt cây tổng quát cho cây nhị phân.

Ta có thể xây dựng được định nghĩa của phép duyệt cây tổng quát T như sau:

Duyệt theo thứ tự trước

1. Nếu T rỗng thì không làm gì
2. Nếu T khác rỗng thì:

   Thăm gốc của T

   Duyệt các cây con thứ nhất T_1 của gốc của cây T theo thứ tự trước

   Duyệt các cây con còn lại T_2, T_3,..., T_n của gốc T theo thứ tự trước.

Duyệt theo thứ tự giữa

1. Nếu T rỗng thì không làm gì
2. Nếu T khác rỗng thì:

   Duyệt cây con thứ nhất T_1 của gốc của cây T theo thứ tự giữa

   Thăm gốc của cây T

   Duyệt các cây con còn lại T_2, T_3,..., T_n của gốc của cây T theo thứ tự giữa.

Duyệt theo thứ tự sau

1. Nếu T rỗng thì không làm gì
2. Nếu T khác rỗng thì:

   Duyệt cây con thứ nhất T_1 của gốc của cây T theo thứ tự sau

   Duyệt các cây con còn lại T_2, T_3,...,T_n của gốc của cây T theo thứ tự sau

   Thăm gốc của cây T.

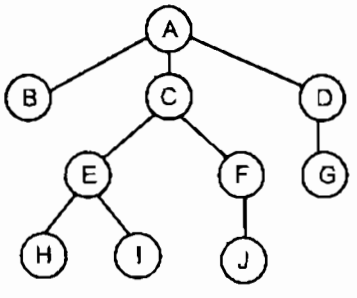
Ví dụ với cây ở hình 5.15 thì dãy tên các nút được thăm sẽ là:

Thứ tự trước: A B C E H I F J D G

Thứ tự giữa: B A H E I C J F G D

Thứ tự sau: B H I E J F C G D A

Bây giờ ta dựng cây nhị phân tương đương với cây tổng quát ở hình 5.15.



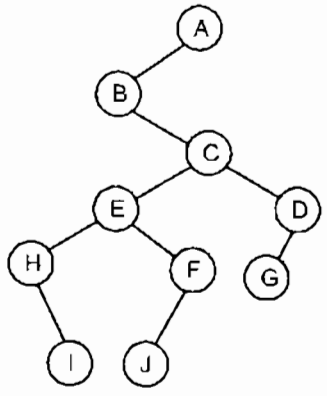
Hình 5.15: Cây tổng quát thể hiện quá trình duyệt

Dãy các nút được thăm khi duyệt nó theo phép duyệt của cây nhị phân:

Thứ tự trước: A B C E H I F J D G

Thứ tự giữa: B H I E J F C G D A

Thứ tự sau: I H JF E G D C B A



Hình 5.16: Cây nhị phân tương đương

Nhận xét: Với thứ tự trước, phép duyệt cây tổng quát và phép duyệt cây nhị phân tương đương của nó đều cho một dãy tên như nhau. Phép duyệt cây tổng quát theo thứ tự sao cho dãy tên giống như dãy tên các nút được duyệt theo thứ tự giữa trong phép duyệt cây nhị phân. Còn phép duyệt cây tổng quát theo thứ tự giữa thì cho dãy tên không giống bất kỳ dãy nào đối với cây nhị phân tương đương. Do đó đối với cây tổng quát, nếu định nghĩa phép duyệt như trên người ta thường chỉ nêu hai phép duyệt theo thứ tự trước và phép duyệt theo thứ tự sau.

4. TÌM KIẾM TRÊN CÂY NHỊ PHÂN

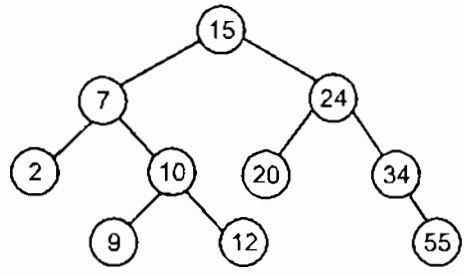
Cây nhị phân được sử dụng vào nhiều mục đích khác nhau. Tuy nhiên việc sử dụng cây nhị phân để lưu giữ và tìm kiếm thông tin vẫn là một trong những áp dụng quan trọng nhất của cây nhị phân. Trong phần này chúng ta sẽ nghiên cứu một lớp cây nhị phân đặc biệt phục vụ cho việc tìm kiếm thông tin, đó là cây nhị phân tìm kiếm.

Trong thực tế, một lớp đối tượng nào đó có thể được mô tả bởi một kiểu bản ghi, các trường của bản ghi biểu diễn các thuộc tính của đối tượng. Trong bài toán tìm kiếm thông tin, ta thường quan tâm đến một nhóm các thuộc tính nào đó của đối tượng mà thuộc tính này hoàn toàn xác định được đối tượng. Chúng ta gọi các thuộc tính này là khoá. Như vậy, khoá là một nhóm các thuộc tính của một lớp đối tượng sao cho hai đối tượng khác nhau phải có giá trị khác nhau trên nhóm thuộc tính đó.

4.1. Định nghĩa

Cây nhị phân tìm kiếm (CNPTK) là cây nhị phân hoặc rỗng hoặc không rỗng thì phải thoả mãn đồng thời các điều kiện sau:

- Khoá của các nút thuộc cây con trái nhỏ hơn khoá nút gốc
- Khoá của nút gốc nhỏ hơn khoá của các nút thuộc cây con phải của nút gốc
- Cây con trái và cây con phải của gốc cũng là cây nhị phân tìm kiếm.



Hình 5.17: Cây nhị phân tìm kiếm

Hình 5.17 biểu diễn một cây nhị phân tìm kiếm, trong đó khoá của các đỉnh là các số nguyên.

4.2. Cài đặt cây nhị phân tìm kiếm

Mỗi nút trên cây tìm kiếm nhị phân có dạng

Left	infor	Right

Trong đó trường Left: con trỏ chỉ tới cây con trái

Right: con trỏ chỉ tới cây con phải

infor: chứa thông tin của nút

Giả sử dữ liệu trên mỗi nút của cây có kiểu dữ liệu là ***Item***, khi đó cấu trúc dữ liệu của CNPTK được định nghĩa như sau:

```
struct Node
{
  Item infor;
  struct Node *Left, *Right;
};
struct Node *Root;
```

Tiếp theo ta nghiên cứu các phép toán trên cây nhị phân tìm kiếm.

4.3. Các thao tác cơ bản trên cây nhị phân tìm kiếm

4.3.1. Tìm kiếm

Tìm kiếm trên cây là một trong các phép toán quan trọng nhất đối với cây nhị phân tìm kiếm. Ta xét bài toán sau

Bài toán: Giả sử mỗi nút trên cây nhị phân tìm kiếm là một bản ghi, biến con trỏ Root chỉ tới gốc của cây và X là khoá cho trước. Vấn đề đặt ra là, tìm xem trên cây có chứa nút với khoá X hay không.

**** Giải thuật đệ qui***

```
struct Node *search (struct Node *Root; Item X)
/* Hàm search trả về con trỏ trỏ tới nút có khoá bằng X, ngược lại trả về NULL*/
{
  if (Root = = NULL) return NULL;
  else
        if (X < Root->infor) return search(Root->Left, X);
        else
              if (X > Root->infor) return search(Root->Right, X);
              else return Root;
}
```

**** Giải thuật lặp***

Trong thủ tục này, ta sẽ sử dụng biến địa phương *found* có kiểu int để điều khiển vòng lặp, nó có giá trị ban đầu là 0. Nếu tìm kiếm thành công thì *found* nhận giá trị 1, vòng lặp kết thúc và hàm *search*() trả về con trỏ trỏ đến nút có trường khoá bằng X. Còn nếu không tìm thấy thì giá trị của *found* vẫn là 0 và hàm *search*() trả về NULL.

```
struct Node *search (struct Node *Root, Item X)
{
        int found=0;
        struct Node *p;
        p = Root;
        while (p != NULL && found==0)
```

```
        {
                if (X > p->infor)
                p = p->Right;
                else
                        if (X < p->infor)
                        p = p->Left;
                        else     found = 1;
        }
        return p;
}
```

4.3.2. Đi qua CNPTK

Như ta đã biết CNPTK cũng là cây nhị phân nên các phép duyệt trên cây nhị phân cũng vẫn đúng trên CNPTK. Chỉ có lưu ý nhỏ là khi duyệt theo thứ tự giữa thì được dãy khoá theo thứ tự tăng dần.

4.3.3. Chèn một nút vào CNPTK

Việc thêm một nút có trường khoá bằng X vào cây phải đảm bảo điều kiện ràng buộc của CNPTK. Ta có thể thêm vào nhiều chỗ khác nhau trên cây, nhưng nếu thêm vào một nút lá sẽ là tiện lợi nhất, do ta có thể thực hiện quá trình tương tự như thao tác tìm kiếm. Khi kết thúc việc tìm kiếm cũng chính là lúc tìm được chỗ cần chèn.

**** Giải thuật lặp***

Trong thủ tục này ta sử dụng biến con trỏ địa phương Q chạy trên các nút của cây bắt đầu từ gốc. Khi đang ở một nút nào đó, Q sẽ xuống nút con trái (hay phải) tuỳ theo khoá ở nút gốc lớn hơn (hay nhỏ hơn) khoá X.

Ở tại một nút nào đó khi P muốn xuống nút con trái (phải) thì phải kiểm tra xem nút này có nút con trái (phải) không. Nếu có thì tiếp tục xuống, ngược lại thì treo nút mới vào bên trái (phải) nút đó. Điều kiện Q = NULL sẽ kết thúc vòng lặp. Quá trình này được lại lặp khi có đỉnh mới được chèn vào.

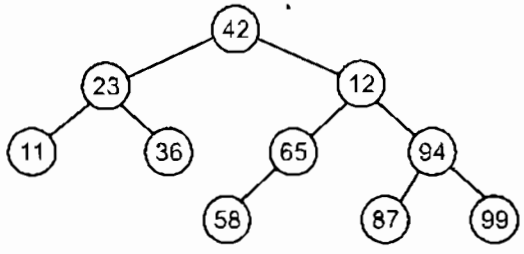
```
/* Hàm insert() bổ sung thêm nút có khóa X vào cây, hàm trả về 1 nếu
bổ sung thành công, ngược lại hàm trả về 0 - trường hợp nút có khóa
X đã có trên cây*/
int Insert (struct Node **Root, Item X)
{
        struct Node *P, *Q;
        P = new Node;
        P->infor = X;
        P->Left = NULL,
        P->Right = NULL;
        if (*Root = = NULL)
        {
                *Root = P;
                return 1;
        }
        else
        {       Q = *Root;
                while (Q != NULL)
                if (X < Q->infor)
                {
                        if (Q->Left != NULL) Q = Q->Left;
                        else    { Q->Left = P;
                                  return 1;
                                }
                }

        else
                if (X > Q->infor)
                {
                        if (Q->Right != NULL) Q = Q->Right;
                        else {
                                Q->Right = P;
                                return 1;
                              }
```

```
                }
                else return 0;
            }
    }
```

Nhận xét:

Để dựng được CNPTK ứng với một dãy khoá đưa vào bằng cách liên tục bổ các nút ứng với từng khoá, bắt đầu từ cây rỗng. Ban đầu phải dựng lên cây với nút gốc là khoá đầu tiên sau đó đối với các khoá tiếp theo, tìm trên cây không có thì bổ sung vào.

Ví dụ với dãy khoá: 42 23 74 11 65 58 94 36 99 87 thì cây nhị phân tìm kiếm dựng được sẽ có dạng ở hình 5.18.



Hình 5.18: Một cây nhị phân tìm kiếm

4.3.4. Loại bỏ nút trên cây nhị phân tìm kiếm

Đối lập với phép toán chèn vào là phép toán loại bỏ. Chúng ta cần phải loại bỏ khỏi CNPTK một đỉnh có khoá X (ta gọi tắt là nút X) cho trước, sao cho việc huỷ một nút ra khỏi cây cũng phải bảo đảm điều kiện ràng buộc của CNPTK.

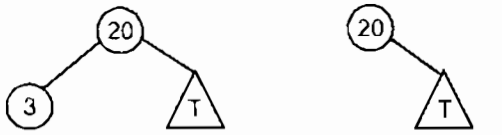
Có ba trường hợp khi huỷ một nút X có thể xảy ra:

- X là nút lá

- X là nút nửa lá (chỉ có một con trái hoặc con phải)

- X có đủ hai con (trường hợp tổng quát)

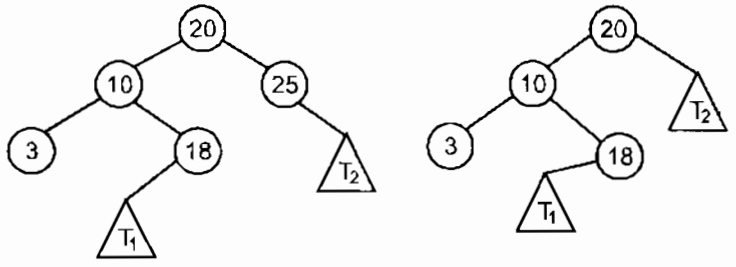
Trường hợp thứ nhất: chỉ đơn giản huỷ nút X vì nó không liên quan đến phần tử nào khác.



Cây trước khi xóa Cây sau khi xóa

Hình 5.19: Xóa nút lá trên cây

Trường hợp thứ hai: Trước khi xoá nút X cần móc nối cha của X với nút con (nút con trái hoặc nút con phải) của nó.

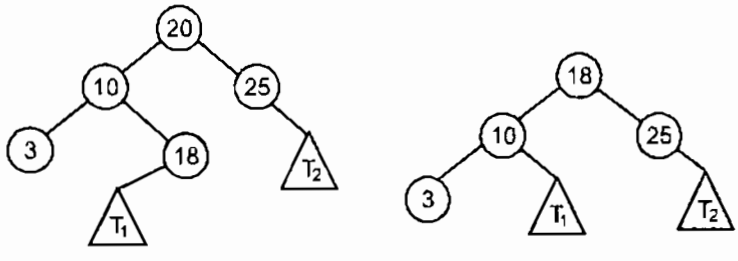


a. Cây trước khi xóa b. Cây sau khi xóa đỉnh (25)

Hình 5.20: Xóa nút nửa lá

Trường hợp tổng quát: khi nút bị loại bỏ có cả cây con trái và cây con phải, thì nút thay thế nó hoặc là nút ứng với khoá nhỏ hơn ngay sát trước nó (nút cực phải của cây con trái nó) hoặc nút ứng với khoá lớn hơn ngay sát sau nó (nút cực trái của cây con phải nó). Như vậy ta sẽ phải thay đổi một số mối nối ở các nút:

- Nút cha của nút bị loại bỏ
- Nút được chọn làm nút thay thế
- Nút cha của nút được chọn làm nút thay thế.

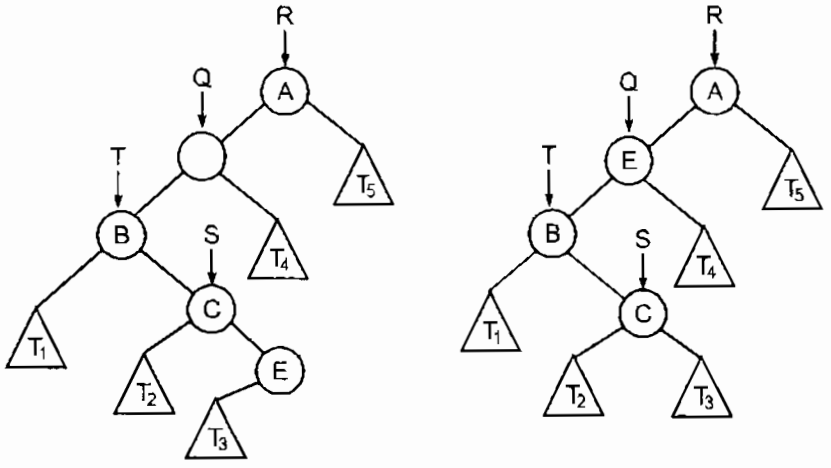


1. Cây trước khi xóa đỉnh 20 2. Cây sau khi xóa đỉnh 20

Hình 5.21: Xóa nút đầy đủ

Ở đây ta chọn nút thay thế nút bị xoá là nút cực phải của cây con trái (nút 18).

Sau đây là giải thuật thực hiện việc loại bỏ một nút trỏ bởi Q. Ban đầu Q chính là nối trái hoặc nối phải của một nút R trên cây nhị phân tìm kiếm, mà ta giả sử đã biết rồi.



1. Cây trước khi xóa nút trỏ bởi Q 2. Cây sau khi xóa nút trỏ bởi Q

Hình 5.22: Xóa nút trỏ bởi con trỏ Q

```
void Del (struct Node *Q)        //xoá nút trỏ bởi Q
{
        struct Node *T,*S, *P;
        P = Q;  //Xử lý trường hợp nút lá và nút nửa lá
        if (P->Left == NULL)
        {
                Q=P->Right; //R^.Left = P^.Right
                delete P;
        }
        else
                if (P->Right == NULL)
                {       Q = P->Left; delete P;
                }
                else    //Xử lý trường hợp tổng quát
```

```
        {
            T = P->Left;
            if (T->Right == NULL)
            {
                T->Right = P->Right;
                Q = T;
                delete P;
            }
            else
            {
                S = T->Right;
                //Tìm nút thay thế, là nút cực phải của cây
                while (S->Right != NULL)
                {
                    T = S;
                    S = T->Right;
                }
                S->Right = P->Right;
                T->Right = S->Left;
                S->Left = P->Left;
                Q = S;
                delete P;
        }
    }
}
```

Thủ tục xoá trường dữ liệu bằng X

- Tìm đến nút có trường dữ liệu bằng X
- Áp dụng thủ tục Delete để xoá

Sau đây chúng ta sẽ viết thủ tục loại khỏi cây gốc Root đỉnh có khoá X cho trước. Đó là thủ tục đệ quy, nó tìm ra đỉnh có khoá X, sau đó áp dụng thủ tục Del để loại đỉnh đó ra khỏi cây.

```
void Delete (Item X)
{
  if (Root != NULL)
```

```
            if (x < Root->infor) Delete(Root->Left, X)
            else if (X > Root->infor) Delete(Root->Right, X)
            else Dellete (Root);
}
```

Nhận xét:

Việc huỷ toàn bộ cây có thể thực hiện thông qua thao tác duyệt cây theo thứ sau. Nghĩa là ta sẽ huỷ cây con trái, cây con phải rồi mới huỷ nút gốc.

```
void RemoveTree ()
{
  if (Root != NULL)
  {
            RemoveTree(Root->Left);
            RemoveTree(Root->Right);
            delete Root;
  }
}
```

4.4. Thời gian thực hiện các phép toán trên CNPTK

Trong mục này ta sẽ đánh giá thời gian để thực hiện các phép toán trên CNPTK. Ta có nhận xét rằng, thời gian thực các phép tìm kiếm là số phép so sánh giá trị khoá X cho trước với khoá của các nút nằm trên đường đi từ gốc tới nút nào đó trên cây. Do đó, thời gian thực hiện các phép tìm kiếm, bổ sung và loại bỏ là độ dài đường đi từ gốc tới một nút nào đó trên cây.

Với giải thuật tìm kiếm nêu trên, ta thấy dạng cây nhị phân tìm kiếm dựng được hoàn toàn phụ thuộc vào dãy khoá đưa vào. Như vậy nghĩa là trong quá trình xử lý động ta không thể biết trước được cây sẽ phát triển ra sao, hình dạng của nó sẽ như thế nào.

Trong trường hợp nó là một cây nhị phân hoàn chỉnh (ta gọi là cân đối ngay cả khi nó chưa đầy đủ) thì chiều cao của nó là $\lceil \log_2(n+1) \rceil$, nên chi phí tìm kiếm có cấp độ là $O(\log_2 n)$.

Trong trường hợp nó là một cây nhị phân suy biến thành một danh sách tuyến tính (khi mà mỗi nút chỉ có một con trừ nút lá). Lúc đó các thao tác trên cây sẽ có độ phức tạp là O(n).

Người ta chứng minh được rằng số lượng trung bình các phép so sánh trong tìm kiếm trên cây nhị phân tìm kiếm là:

$$C_{tb} \approx 1{,}386 \log_2 n$$

Do đó cấp độ lớn của thời gian thực hiện trung bình các phép toán cũng chỉ là $O(\log_2 n)$.

5. CÂY CÂN BẰNG (AVL TREE)

Giả sử ta có một tập hợp dữ liệu nào đó. Vấn đề đặt ra là, ta phải tổ chức các dữ liệu đó như thế nào sao cho việc cập nhật thông tin (tìm kiếm, bổ sung và loại bỏ) đạt hiệu quả nhất. Với các kiểu dữ liệu này người ta có tổ chức trên cây nhị phân tìm kiếm. Khi đó thời gian trung bình thực hiện các phép toán là $O(\log_2 n)$. Trong nhiều áp dụng chúng ta cần thường xuyên thực hiện các phép toán bổ sung và loại bỏ khỏi CNPTK. Điều này có thể dẫn đến trường hợp cây suy biến thành danh sách tuyến tính. Đối với những cây này, việc thực hiện các phép toán kém hiệu quả do chi phí thời gian thực hiện là O(n). Để khắc phục nhược điểm này ta tổ chức dữ liệu trên một lớp cây đặc biệt sao cho thời gian thực hiện các phép toán luôn là $O(\log_2 n)$.

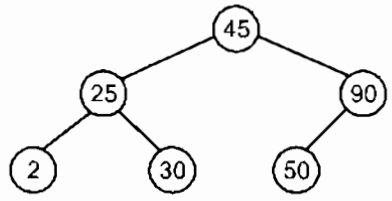
5.1. Cây cân bằng hoàn toàn

Định nghĩa: Cây cân bằng hoàn toàn (CCBHT) là cây nhị phân tìm kiếm mà tại mỗi nút của nó, số nút của cây con trái và cây con phải không chênh lệch nhau quá một.

Một cây rất khó đạt được trạng thái cân bằng hoàn toàn và cũng rất dễ mất cân bằng vì khi thêm hay huỷ các nút trên cây có thể làm mất cân bằng (xác suất rất lớn), chi phí cân bằng lại cây là rất lớn vì phải thao tác trên toàn bộ cây.

Tuy nhiên nếu cây cân đối thì việc tìm kiếm sẽ rất nhanh. Đối với CCBHT, trong trường hợp xấu nhất ta cũng chỉ phải tìm qua log2n phần tử (n là số nút trên cây).

Hình 5.23 là ví dụ về một CCBHT.



Hình 5.23: Cây cân bằng hoàn toàn

CCBHT có n nút, có chiều cao h = $\log_2 n$. Đây chính là lý do cho phép đảm bảo khả năng tìm kiếm nhanh trên cấu trúc dữ liệu này.

Do CCBHT là một cấu trúc kém ổn định nên trong thực tế không thể sử dụng. Nhưng ưu điểm của nó lại rất quan trọng. Vì vậy, cần đưa ra một cấu trúc dữ liệu khác có đặc tính giống CCBHT nhưng ổn định hơn.

Như vậy, cần tìm cách tổ chức một cây đạt trạng thái cân bằng yếu hơn và việc cân bằng lại chỉ xảy ra ở phạm vi cục bộ nhưng vẫn đảm bảo chi phí cho thao tác tìm kiếm đạt ở mức $O(\log_2 n)$.

5.2. Cây cân bằng

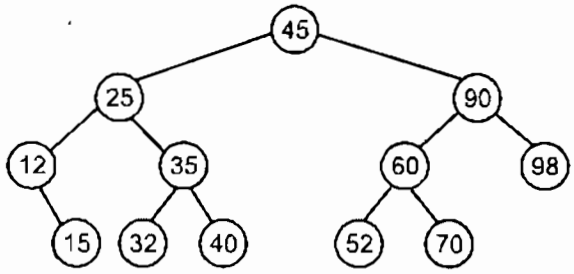
Năm 1962, P.M.ADELSON - VELSKI và E.LANDIS đã mở đầu cách giải quyết này bằng cách đưa ra dạng cây cân đối mới mà sau này mang tên của họ, đó là cây nhị phân cân đối AVL. Chúng ta sẽ dùng thuật ngữ cây AVL thay cho cây cân bằng.

Từ khi được giới thiệu, cây AVL đã nhanh chóng được ứng dụng trong nhiều bài toán khác nhau. Vì vậy, nó mau chóng trở nên thịnh hành và thu hút nhiều nghiên cứu. Từ cây AVL, người ta đã phát triển thêm nhiều loại cấu trúc dữ liệu hữu dụng khác như cây đỏ - đen, B - Tree, v.v...

5.2.1. Định nghĩa

Cây AVL là cây nhị phân tìm kiếm mà tại mỗi nút của nó độ cao của cây con trái và của cây con phải chênh lệch nhau không quá một.

Hình 5.24 là ví dụ cây cân bằng.



Hình 5.24: Cây AVL

Dễ dàng thấy CCBHT là cây cân bằng. Điều này ngược lại không đúng. Tuy nhiên, cây AVL là cấu trúc dữ liệu ổn định hơn hẳn CCBHT.

5.2.2. Chiều cao của cây AVL

Một vấn đề quan trọng, như đã đề cập trong phần trước, là ta phải khẳng định cây AVL n nút, phải có chiều cao khoảng $\log_2 n$.

Để đánh giá chính xác về chiều cao của cây AVL, ta xét bài toán: cây AVL có chiều cao h sẽ có tối thiểu bao nhiêu nút?

Gọi N(h) số nút tối thiểu của cây AVL có chiều cao h.

Ta có N(0) = 0, N(1) = 1 và N(2) = 2.

Cây AVL tối thiểu có chiều cao h sẽ có cây con AVL tối thiểu chiều cao h-1 và cây con AVL tối thiểu chiều cao h-2. Như vậy:

$N(h) = 1 + N(h - 1) + N(h - 2)$ (1)

Ta lại có: $N(h - 1) > N(h - 2)$

Nên từ (1) suy ra:

$N(h) > 2N(h - 2)$

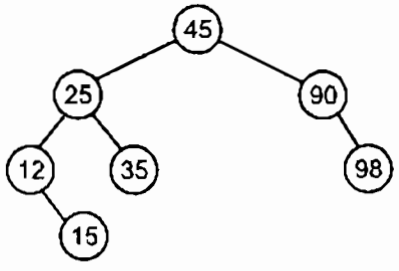
$N(h) > 2^2 N(h - 4)$

...

$N(h) > 2^i N(h - 2i)$

$\Rightarrow N(h) > 2^{h/2-1}$

$\Rightarrow h < 2\log_2(N(h)) + 2$



Hình 5.25: Cây AVL tối thiểu

Như vậy, cây AVL có chiều cao $O(\log_2 n)$.

Ví dụ: cây AVL tối thiểu có chiều cao h = 4 (hình 5.25).

5.2.3. Chỉ số cân bằng của một nút

Định nghĩa: chỉ số cân bằng của một nút là hiệu của chiều cao cây con phải và cây con trái của nó.

Đối với cây AVL chỉ số cân bằng của mỗi nút có thể nhận một trong ba giá trị sau đây:

0: độ cao cây con trái bằng độ cao cây con phải

1: cây con phải có độ cao lớn hơn độ cao cây con trái là 1 (lệch phải)

-1: cây con trái có độ cao lớn hơn độ cao cây con phải là 1 (lệch trái).

5.2.4. Biểu diễn cây AVL

Để khảo sát cây cân bằng, ta cần lưu thêm thông tin về chỉ số cân bằng tại mỗi nút. Khi đó cấu trúc dữ liệu của cây cân bằng được khai báo như sau:

```
struct Node
{
        Item infor;
        struct Node *Left, *Right;
        int bal;
};
struct Node *Root;
```

Trong khai báo trên ta bổ sung thêm trường bal (balance: cân bằng), *bal=0* hai cây con cao bằng nhau, *bal=1* cao bên phải, *bal=-1* cao bên trái.

Sau đây chúng ta sẽ xét các phép toán bổ sung và loại bỏ trên cây AVL.

5.2.5. Bổ sung trên cây AVL

Khi bổ sung nút mới với khoá X cho trước được thực hiện bằng cách sau:

- Áp dụng giải thuật bổ sung vào cây nhị phân tìm kiếm

- Cân bằng lại các đỉnh mà tại đó tính cân bằng bị phá vỡ (độ cao của hai cây con khác nhau 2).

Xét các trường hợp sau

Trường hợp 1: Cây lệch trái (lệch phải) sau khi bổ sung vào cây con phải (trái) cây cân bằng.

Trường hợp 2: Hai cây con có độ cao bằng nhau sau khi bổ sung thì cây lệch trái hoặc lệch phải.

Trường hợp 3: Cây con trái (phải) cao hơn cây con phải (trái) 1, sau khi bổ sung vào cây con trái (phải) thì hai cây này có độ cao chênh nhau là 2. Như vậy, tính cân bằng bị phá vỡ.

Đối với các trường hợp 1 (TH1) và trường hợp 2 (TH2) khi bổ sung nút mới thì tính cân bằng không bị phá vỡ nên ta chỉ cần chỉnh lại các hệ số cân bằng ở nút đang xét và ở các nút tiền bối của nó.

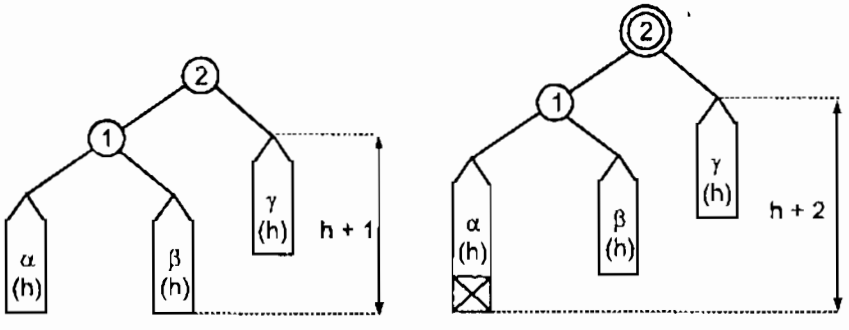
Đối với trường hợp 3 (TH3) ta phải sửa lại cây con mà ta đang xét là nút gốc (ta sẽ gọi là nút bất thường).

Qui ước:

(2) chỉ nút bất thường ứng với khoá bằng 2

⊠ chỉ nút mới bổ sung

α(h) chỉ cây con α có chiều cao h

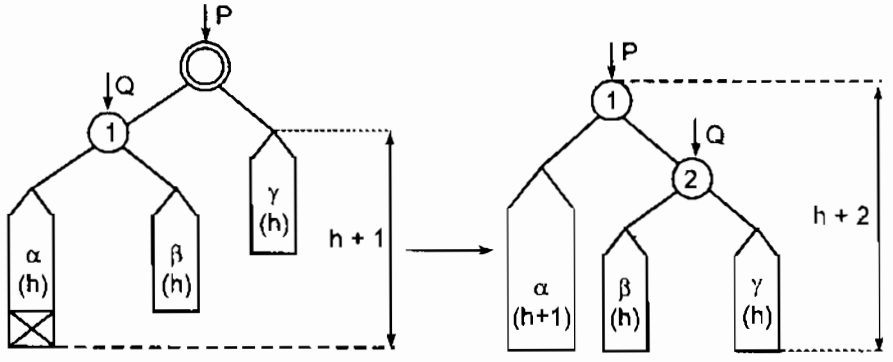


a) Trước khi bổ sung b) Sau khi bổ sung cây mất cân đối

Hình 5.26: Bổ sung vào cây AVL

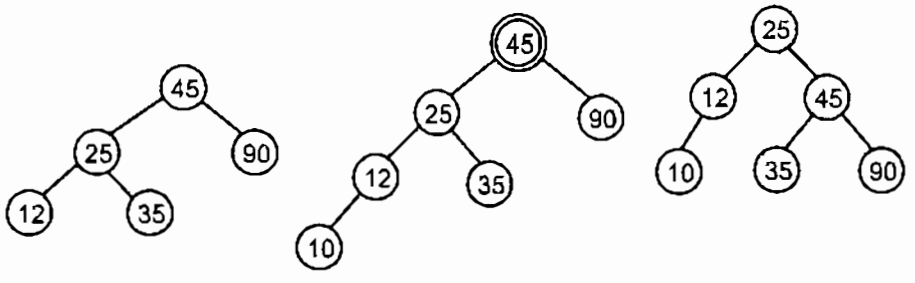
- *Trường hợp 3.1*: Nút bổ sung làm tăng chiều cao cây con trái của nút con trái nút bất thường

Đối với trường hợp này để tái cân đối ta phải thực hiện phép quay từ trái sang phải để đưa nút (1) lên vị trí gốc cây con, nút (2) sẽ trở thành con phải của nó và β được gắn vào thành con trái của (2). Người ta gọi phép quay này là phép quay đơn (single rotation) hay ta còn gọi là phép quay phải.



Hình 5.27: Quay phải cây P

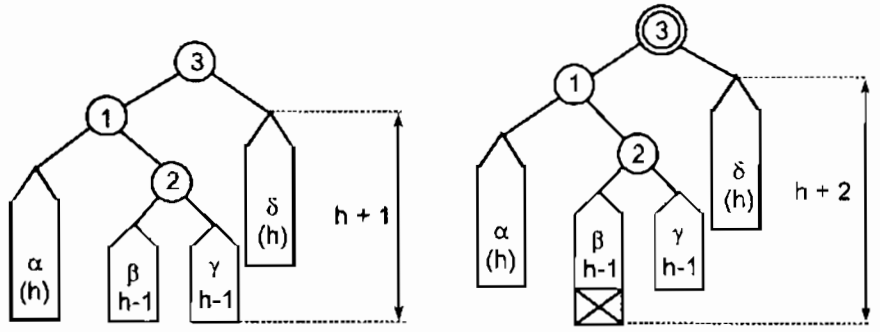
Ví dụ: Cho CNPTK hình 5.28.



a) Cây ban đầu b) Bổ sung thêm nút (10) c) Trái cân đối bằng phép quay đơn

Hình 5.28: Phép quay đơn

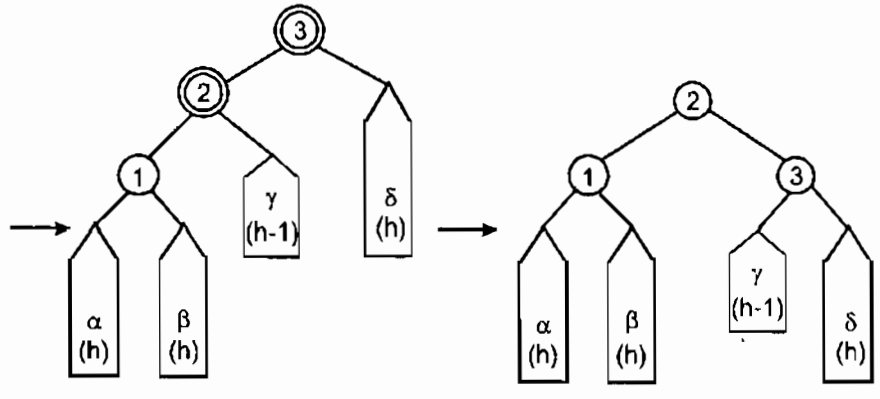
- *Trường hợp 3.2*: Nút mới bổ sung làm tăng chiều cao cây con phải của nút con trái nút bất thường.



a) Trước khi bổ sung b) Sau khi bổ sung cây mất cân đối

Hình 5.29: Bổ sung vào cây AVL

Đối với trường hợp này để tái cân đối ta sử dụng phép quay kép (double rotation), đó là việc phối hợp hai phép quay đơn: quay trái đối với cây con trái ((1), (2)) và quay phải đối với cây ((3), (2)) như hình 5.30.

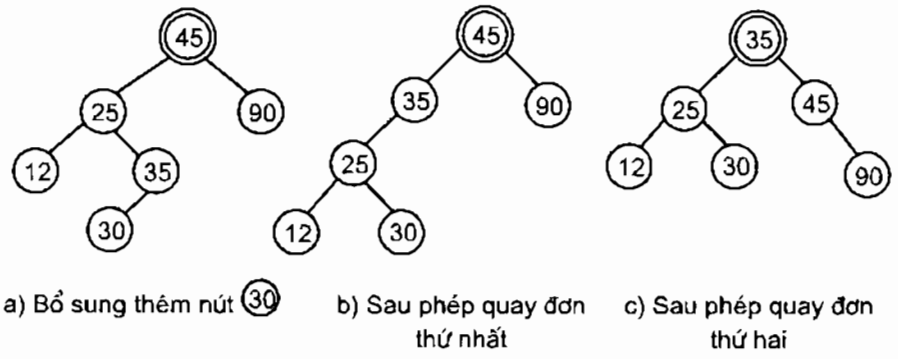


a) Sau phép quay đơn thứ nhất quay trái

b) Sau phép quay đơn thứ hai quay phải

Hình 5.30: Phép quay kép

Ví dụ: Cho cây như hình 5.31.



a) Bổ sung thêm nút 30 b) Sau phép quay đơn thứ nhất c) Sau phép quay đơn thứ hai

Hình 5.31: Minh họa phép quay kép

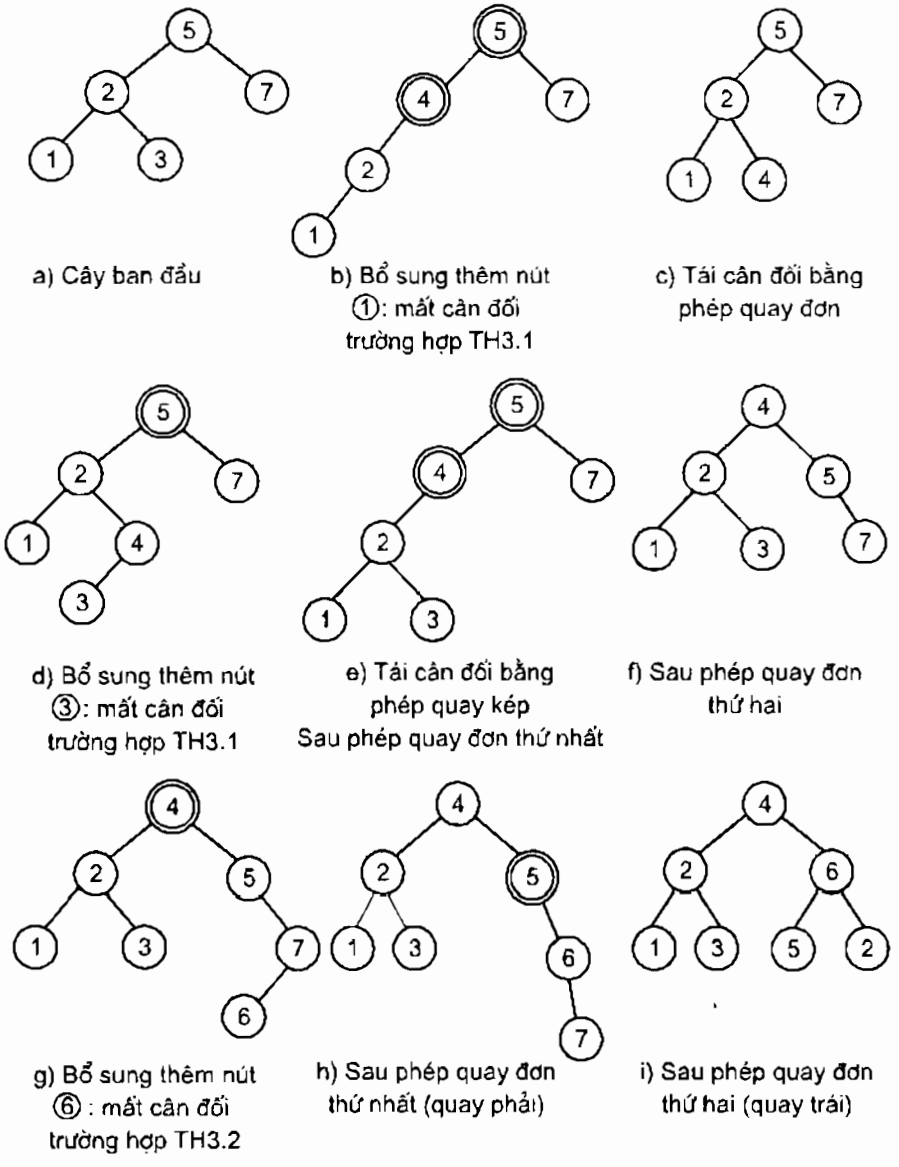
Nhận xét:

Sau khi thực hiện phép quay để tái cân đối cây con mà nút gốc là nút bất thường, chiều cao của các cây con đó vẫn giữ nguyên như trước lúc bổ sung, nghĩa là phép quay không làm ảnh hưởng tới chiều cao các cây có liên quan tới cây con này.

Trong quá trình thực hiện các phép quay, tính chất của cây nhị phân tìm kiếm luôn luôn được đảm bảo.

Đối với các trường hợp khi bổ sung thêm nút mới làm tăng chiều cao cây con phải của nút con phải và nút mới bổ sung làm chiều cao cây con trái của nút con phải nút bất thường, thì ta lần lượt thực hiện theo thứ tự ngược lại tương ứng với trường hợp TH3.1 và TH3.2.

Ví dụ sau đây minh hoạ cụ thể và các phép xử lý tương ứng.



Hình 5.32 : Minh họa các phép xử lý trên cây AVL

5.2.6. Huỷ một nút trên cây AVL

Cũng giống như thao tác thêm một nút, việc huỷ nút X ra khỏi cây AVL thực hiện giống như trên CNPTK. Chỉ sau khi huỷ, nếu tính cân bằng bị phá vỡ thì ta sẽ thực hiện việc cân bằng lại.

Tuy nhiên, việc cân bằng lại trong thao tác huỷ sẽ phức tạp hơn nhiều so với việc cân bằng khi thêm một nút do có thể xảy ra phản ứng dây chuyền.

Nhận xét:

- Thao tác thêm một nút có độ phức tạp O(1)

- Thao tác huỷ một nút có độ phức tạp O(h)

- Với cây cân bằng trung bình hai lần thêm vào cây thì cần một lần cân bằng lại; 5 lần huỷ thì cần một lần cân bằng lại.

- Việc huỷ một nút có thể phải cân bằng dây chuyền các nút từ gốc cho đến phần tử bị huỷ trong khi thêm vào chỉ cần một lần cân bằng cục bộ.

- Độ dài đường tìm kiếm trung bình trong cây cân bằng gần bằng cây cân bằng hoàn toàn, nhưng việc cân bằng lại đơn giản hơn nhiều

- Một cây cân bằng không bao giờ cao hơn 45% cây cân bằng hoàn toàn tương ứng dù số nút trên cây là bao nhiêu.

Trong chương này chúng tôi đã trình bày các kiến thức căn bản về một dạng cấu trúc dữ liệu khá phức tạp, cấu trúc cây. Cây có nhiều dạng như cây tổng quát, cây nhị phân. Cấu trúc lưu trữ cũng khác nhau có thể lưu trữ cây bởi mảng hoặc con trỏ. Mong rằng qua đây bạn đọc nắm bắt được cách thức xây dựng cấu trúc dữ liệu dạng cây cùng với các phép xử lý và các thao tác trên cây, từ đó định ra phương pháp xây dựng phần mềm của mình một cách hiệu quả.

BÀI TẬP CHƯƠNG 5

1. Dựng cây nhị phân biết thứ tự các đỉnh khi duyệt theo:

a. Thứ tự trước: A D F G H K L P Q R W Z

Thứ tự giữa: G F H K D L A W R Q P Z

b. Theo thứ tự sau: F G H D A L P Q R Z W K

Thứ tự giữa: G F H K D L A W R Q P Z

2. Với mỗi biểu thức số học dưới đây, hãy vẽ cây nhị phân biểu diễn biểu thức ấy, rồi dùng các kiểu duyệt để tìm biểu thức tiền tố và hậu tố tương đương.

   a. a/(b - (c - (d - (e - f))))

   b. ((a * (b + c))/(d - (e + f))) * (g/(h/(i * j)))

3. Hãy trình bày các vấn đề sau:

   a. Định nghĩa và đặc điểm của cây nhị phân tìm kiếm

   b. Thao tác thực hiện tốt nhất trong kiểu này

   c. Hạn chế của kiểu này là gì?

4. Xét giải thuật tạo cây nhị phân tìm kiếm. Nếu thứ tự các khoá nhập vào là như sau:

   8 3 5 2 20 11 30 9 18 4

   thì hình ảnh cây được tạo như thế nào?

   Sau đó, nếu huỷ lần lượt các nút theo thứ tự như sau:

   11 20 8

   thì cây sẽ thay đổi như thế nào trong từng bước huỷ, vẽ cây minh hoạ qua từng bước.

5. Áp dụng thuật giải tạo cây AVL để dựng cây với thứ tự các khoá nhập vào như sau: 5 7 2 1 3 6 10 thì hình ảnh cây tạo được như thế nào? Giải thích rõ từng tình huống xảy ra khi thêm từng khoá vào cây và vẽ hình minh hoạ.

   Sau đó, nếu huỷ lần lượt các nút theo thứ tự như sau:

   5 6 7

   thì cây sẽ thay đổi như thế nào trong từng bước huỷ, vẽ sơ đồ và giải thích.

6. Viết các hàm xác định các thông tin của cây nhị phân T

   - Số nút lá

   - Số nút có đúng một cây con

- Số nút có đúng hai cây con
- Số nút có khoá nhỏ hơn X (giả sử T là CNPTK)
- Số nút có khoá lớn hơn X (giả sử T là CNPTK)
- Chiều cao của cây
- In ra tất cả các nút ở mức thứ k của cây
- Kiểm tra xem T có phải là cây cân bằng hoàn toàn không.

7. Xây dựng cấu trúc dữ liệu biểu diễn cây n- phân và tạo sinh cây nhị phân tương ứng với các khoá của cây n - phân.

   Giả sử khoá được lưu trữ chiếm k byte, mỗi con trỏ chiếm 4 byte, vậy dùng cây nhị phân thay cho cây n- phân thì có lợi gì trong việc lưu trữ các khoá.

8. Viết hàm chuyển một cây n- phân thành cây nhị phân.

9. Hãy tìm một ví dụ về một cây AVL có chiều cao là 6 và khi huỷ một nút lá (chỉ cụ thể) việc cân bằng lại lan truyền lên tận gốc của cây. Vẽ ra từng bước của quá trình huỷ và cân bằng lại này.

10. Cài đặt chương trình mô phỏng trực quan các thao tác trên cây nhị phân tìm kiếm.

11. Cài đặt chương trình mô phỏng trực quan các thao tác trên cây AVL

    Viết chương trình cho phép tạo, tra cứu và sửa chữa từ điển Anh - Việt.

12. Thông tin của mỗi nhân viên bao gồm mã nhân viên (kiểu số); tên nhân viên (kiểu chuỗi). Danh sách các nhân viên như sau: (30, Tùng), (35, Lan), (10, Sơn), (25, Nam), (20, Lý), (12, Lan), (40, Tuyết), (50, Cường), (60, Hảo), (55, Ánh), (48, Hương)

    a. Khai báo cấu trúc dữ liệu bằng CNPTK để lưu trữ cac nhân viên trên.

    b. Dựng CNPTK lần lượt các nhân viên này.

    c. Viết hàm cho biết thông tin của các nhân viên có tên là "Lan".

TÀI LIỆU THAM KHẢO

[1] Đỗ Xuân Lôi, *Cấu trúc dữ liệu và giải thuật*, NXB Thống kê, 2000

[2] Đinh Mạnh Tường, *Cấu trúc dữ liệu và giải thuật*, NXB Khoa học và Kỹ thuật, 2001.

[3] Phạm Văn Ất, *Kỹ thuật lập trình C cơ sở và nâng cao*, NXB Khoa học và Kỹ thuật, 1999.

[4] Robert Kruse, C.L.Tondo, Bruce Leung, *Data structures & program design in C.*

[5] J.Courtin, I.Kowarski, *Nhập môn thuật toán và cấu trúc dữ liệu* (Nguyễn Ngọc Kỳ, Lương Chi Mai, Nguyễn Thanh Tùng dịch).

MỤC LỤC

CẤU TRÚC DỮ LIỆU VÀ GIẢI THUẬT

Chịu trách nhiệm xuất bản

NGUYỄN THỊ THU HÀ

Biên tập: NGÔ MỸ HẠNH

BÙI NGỌC KHOA

Trình bày mỹ thuật: NGUYỄN MẠNH HOÀNG

Sửa bản in: BÙI NGỌC KHOA

Thiết kế bìa: TRẦN HỒNG MINH

In 2000 bản, khổ 16 x 24 cm, tại Công ty Cổ phần In Hà Nội

Số đăng ký kế hoạch xuất bản: 365-2009/CXB/6 - 139/TTTT

Số quyết định xuất bản: 182/QĐ-NXB TT&TT ngày 02 tháng 10 năm 2009

In xong và nộp lưu chiểu tháng 10 năm 2009.

NHÀ XUẤT BẢN THÔNG TIN VÀ TRUYỀN THÔNG

Trụ sở chính: 18 Nguyễn Du, Hà Nội

ĐT Biên tập: 04.35772143, 04.35772145 ĐT Phát hành: 04.35772138

E-mail: nxb.tttt@mic.gov.vn Fax: 04.35772037

Website: www.nxbthongtintruyenthong.vn

Chi nhánh TP. Hồ Chí Minh: 8A đường D2, Phường 25, Quận Bình Thạnh, TP. Hồ Chí Minh

Điện thoại: 08.35127750, 35127751 Fax: 08.35127751

E-mail: cnsg.nxbtttt@mic.gov.vn

Chi nhánh TP. Đà Nẵng: 42 Trần Quốc Toản, quận Hải Châu, TP. Đà Nẵng

Điện thoại: 0511.3897467 Fax: 0511.3843359

E-mail: cndn.nxbtttt@mic.gov.vn

MỜI CÁC BẠN ĐÓN ĐỌC

1. NHẬP MÔN TIN HỌC
2. LẬP TRÌNH HƯỚNG ĐỐI TƯỢNG
3. CƠ SỞ DỮ LIỆU
4. PHÂN TÍCH THIẾT KẾ HỆ THỐNG
5. THIẾT KẾ WEB
6. MẠNG CĂN BẢN

SÁCH CỦA NHÀ XUẤT BẢN THÔNG TIN VÀ TRUYỀN THÔNG CÓ BÁN TẠI:

1. **Nhà sách Tiền phong**
   175 Nguyễn Thái Học, Hà Nội
2. **Nhà sách Nguyễn Văn Cừ**
   36 Xuân Thủy, Cầu Giấy, Hà Nội
3. **Nhà sách Minh Châu**
   Số 10 và 14/40 Tạ Quang Bửu, Hai Bà Trưng, Hà Nội
4. **Nhà sách Bách Khoa**
   Số 1, Đường Giải Phóng, Hà Nội
   86 - 107 Tô Hiến Thành, Quận 10, TP. HCM
5. **Nhà sách Thăng Long**
   2 Bis Nguyễn Thị Minh Khai, Quận 1, TP. HCM

Giá: 30.000đ