TRƯỜNG ĐẠI HỌC ĐÀ LẠT
KHOA CÔNG NGHỆ THÔNG TIN

NGUYỄN THỊ THANH BÌNH

NGUYỄN VĂN PHÚC

GIÁO TRÌNH

CẤU TRÚC DỮ LIỆU VÀ THUẬT GIẢI 2

Dành cho sinh viên ngành công nghệ thông tin

Đà Lạt 2010

LỜI NÓI ĐẦU

Để đáp ứng nhu cầu học tập của các bạn sinh viên, nhất là sinh viên chuyên ngành công nghệ thông tin, Khoa Công Nghệ Thông Tin Trường Đại Học Đà Lạt chúng tôi đã tiến hành biên soạn các giáo trình, bài giảng chính trong chương trình học

Giáo trình này được soạn theo đề cương chi tiết môn Cấu Trúc Dữ Liệu Và Thuật Giải 2 của Khoa Công nghệ Thông tin, trường Đại học Đà Lạt. Mục tiêu của giáo trình nhằm giúp các bạn sinh viên chuyên ngành có một tài liệu cô đọng dùng làm tài liệu học tập.

Nội dung giáo trình gồm 4 chương sau:

Chương 1: trình bày cấu trúc dữ liệu cây, trong đó nhấn mạnh về cấu trúc dữ liệu cây nhị phân tìm kiếm BST và cây nhị phân tìm kiếm cân bằng AVL cùng các phép toán trên nó.

Chương 2: trình bày về đồ thị, các cấu trúc dữ liệu dùng biểu diễn đồ thị và một số bài toán trên đồ thị.

Chương 3: trình bày cấu trúc dữ liệu bảng băm, các hàm băm, cách tổ dữ liệu trên bảng băm nhằm phục vụ cho bài toán tìm kiếm được hiệu quả.

Chương 4: giới thiệu về một số phương pháp thiết kế giải thuật cơ bản giúp sinh viên bước đầu làm quen với một số phương pháp thiết kế giải thuật.

Mặc dù đã rất cố gắng nhiều trong quá trình biên soạn giáo trình, xong không khỏi còn nhiều thiếu sót và hạn chế. Rất mong nhận được sự đóng góp ý kiến quý báu của sinh viên và các bạn đọc để giáo trình ngày một hoàn thiện hơn.

Đà Lạt, ngày 30 tháng 08 năm 2010

Mục lục

Chương I

Cây

Mục tiêu

Sau khi học xong chương này, sinh viên phải:

- Nắm vững khái niệm về cây (trees).
- Cài đặt được cây và thực hiện các phép toán trên cây.

Kiến thức cơ bản cần thiết

Để học tốt chương này, sinh viên phải nắm vững kỹ năng lập trình căn bản như:

- Kiểu con trỏ (pointer)
- Các cấu trúc điều khiển, lệnh vòng lặp.
- Lập trình theo từng module (chương trình con) và cách gọi chương trình con đó.
- Lập trình đệ qui và gọi đệ qui.
- Kiểu dữ liệu trừu tượng danh sách

Nội dung

Trong chương này chúng ta sẽ nghiên cứu các vấn đề sau:

- Các thuật ngữ cơ bản.
- Kiểu dữ liệu trừu tượng Cây
- Cây nhị phân
- Cây tìm kiếm nhị phân
- Cây nhị phân tìm kiếm cân bằng AVL

I. Các thuật ngữ cơ bản trên cây

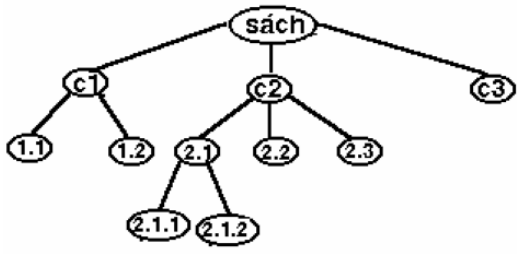
Cây là một tập hợp các phần tử gọi là nút (nodes) trong đó có một nút được phân biệt gọi là nút gốc (root). Trên tập hợp các nút này có một quan hệ, gọi là mối quan hệ cha - con (parenthood), để xác định hệ thống cấu trúc trên các nút. Mỗi nút, trừ nút gốc, có duy nhất một nút cha. Một nút có thể có nhiều nút con hoặc không có nút con nào. Mỗi nút biểu diễn một phần tử trong tập hợp đang xét và nó có thể có một kiểu nào đó bất kỳ, thường ta biểu diễn nút bằng một kí tự, một chuỗi hoặc một số ghi trong vòng tròn. Mối quan hệ cha con được biểu diễn theo qui ước nút cha ở dòng trên nút con ở dòng dưới và được nối bởi một đoạn thẳng. Một cách hình thức ta có thể định nghĩa cây một cách đệ qui như sau:

1. Định nghĩa

- Một nút đơn độc là một cây. Nút này cũng chính là nút gốc của cây.
- Giả sử ta có n là một nút đơn độc và k cây T_1,.., T_k với các nút gốc tương ứng là n_1,.., n_k thì có thể xây dựng một cây mới bằng cách cho nút n là cha của các nút

n_1,.., n_k. Cây mới này có nút gốc là nút n và các cây T_1,.., T_k được gọi là các cây con. Tập rỗng cũng được coi là một cây và gọi là cây rỗng kí hiệu.

Ví dụ: xét mục lục của một quyển sách. Mục lục này có thể xem là một cây



Hình I.1: Cây mục lục sách

Nút gốc là sách, nó có ba cây con có gốc là C1, C2, C3. Cây con thứ 3 có gốc C3 là một nút đơn độc trong khi đó hai cây con kia (gốc C1 và C2) có các nút con.

Nếu n^1,.., n^k là một chuỗi các nút trên cây sao cho n^i là nút cha của nút n^{i+1}, với i=1..k-1, thì chuỗi này gọi là một đường đi trên cây (hay ngắn gọn là đường đi) từ n^1 đến n^k. Độ dài đường đi được định nghĩa bằng số nút trên đường đi trừ 1. Như vậy độ dài đường đi từ một nút đến chính nó bằng không.

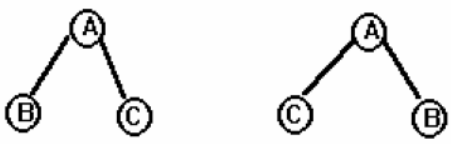
Nếu có đường đi từ nút a đến nút b thì ta nói a là tiền bối (ancestor) của b, còn b gọi là hậu duệ (descendant) của nút a. Rõ ràng một nút vừa là tiền bối vừa là hậu duệ của chính nó. Tiền bối hoặc hậu duệ của một nút khác với chính nó gọi là tiền bối hoặc hậu duệ thực sự. Trên cây nút gốc không có tiền bối thực sự. Một nút không có hậu duệ thực sự gọi là nút lá (leaf). Nút không phải là lá ta còn gọi là nút trung gian (interior). Cây con của một cây là một nút cùng với tất cả các hậu duệ của nó.

Chiều cao của một nút là độ dài đường đi lớn nhất từ nút đó tới lá. Chiều cao của cây là chiều cao của nút gốc. Độ sâu của một nút là độ dài đường đi từ nút gốc đến nút đó. Các nút có cùng một độ sâu i ta gọi là các nút có cùng một mức i. Theo định nghĩa này thì nút gốc ở mức 0, các nút con của nút gốc ở mức 1.

Ví dụ: đối với cây trong hình I.1 ta có nút C2 có chiều cao 2. Cây có chiều cao 3. nút C3 có chiều cao 0. Nút 2.1 có độ sâu 2. Các nút C1,C2,C3 cùng mức 1.

2. Thứ tự các nút trong cây

Nếu ta phân biệt thứ tự các nút con của cùng một nút thì cây gọi là cây có thứ tự, thứ tự qui ước từ trái sang phải. Như vậy, nếu kể thứ tự thì hai cây sau là hai cây khác nhau:



Hình I.2: Cây có thứ tự khác nhau

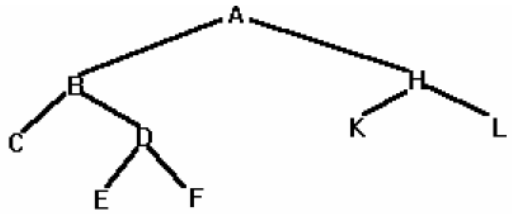
Trong trường hợp ta không phân biệt rõ ràng thứ tự các nút thì ta gọi là cây không có thứ tự. Các nút con cùng một nút cha gọi là các nút anh em ruột (siblings). Quan hệ "trái sang phải" của các anh em ruột có thể mở rộng cho hai nút bất kỳ theo qui tắc: nếu a, b là hai anh em ruột và a bên trái b thì các hậu duệ của a là "bên trái" mọi hậu duệ của b.

3. Các thứ tự duyệt cây quan trọng

Duyệt cây là một qui tắc cho phép đi qua lần lượt tất cả các nút của cây mỗi nút đúng một lần, danh sách liệt kê các nút (tên nút hoặc giá trị chứa bên trong nút) theo thứ tự đi qua gọi là danh sách duyệt cây. Có ba cách duyệt cây quan trọng: Duyệt tiền tự (preorder), duyệt trung tự (inorder), duyệt hậu tự (posorder).

- Cây rỗng thì danh sách duyệt cây là rỗng và nó được coi là biểu thức duyệt tiền tự, trung tự, hậu tự của cây.
- Cây chỉ có một nút thì danh sách duyệt cây gồm chỉ một nút đó và nó được coi là biểu thức duyệt tiền tự, trung tự, hậu tự của cây.
- Ngược lại: giả sử cây T có nút gốc là n và có các cây con là T1,..,Tn thì:
  - Biểu thức duyệt tiền tự của cây T là liệt kê nút n kế tiếp là biểu thức duyệt tiền tự của các cây T1, T2, .., Tn theo thứ tự đó.
  - Biểu thức duyệt trung tự của cây T là biểu thức duyệt trung tự của cây T1 kế tiếp là nút n rồi đến biểu thức duyệt trung tự của các cây T2,.., Tn theo thứ tự đó.
  - Biểu thức duyệt hậu tự của cây T là biểu thức duyệt hậu tự của các cây T1, T2,.., Tn theo thứ tự đó rồi đến nút n.

Ví dụ: cho cây như trong hình I.3



Hình I.3: cây nhị phân

Biểu thức duyệt tiền tự: A B C D E F H K L

trung tự: C B E D F A K H L

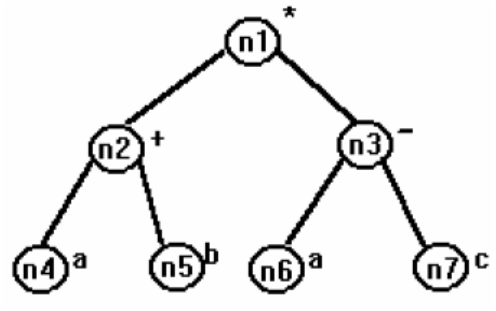
hậu tự: C E F D B K L H A

4. Cây có nhãn và cây biểu thức

Ta thường lưu trữ kết hợp một nhãn (label) hoặc còn gọi là một giá trị (value) với một nút của cây. Như vậy nhãn của một nút không phải là tên nút mà là giá trị được lưu giữ tại nút đó. Nhãn của một nút đôi khi còn được gọi là khóa của nút, tuy nhiên hai khái niệm này là không đồng nhất. Nhãn là giá trị hay nội dung lưu trữ tại nút, còn khoá của nút có thể chỉ là một phần của nội dung lưu trữ này. Chẳng hạn, mỗi nút cây

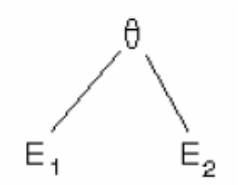
chứa một record về thông tin của sinh viên (mã SV, họ tên, ngày sinh, địa chỉ,...) thì khoá có thể là mã SV hoặc họ tên hoặc ngày sinh tuỳ theo giá trị nào ta đang quan tâm đến trong giải thuật.

Ví dụ: Cây biểu diễn biểu thức (a+b)*(a-c) như trong hình I.4.



Hình I.4: Cây biểu diễn thứ tự (a+b)*(a-c)

- Ở đây n_1, n_2,.., n_7 là các tên nút và *,+,-,a,b,c là các nhãn.
- Qui tắc biểu diễn một biểu thức toán học trên cây như sau:
  - Mỗi nút lá có nhãn biểu diễn cho một toán hạng.
  - Mỗi nút trung gian biểu diễn một toán tử.



Hình I.5: Cây biểu diễn biểu thức E_1 θ E_2

- Giả sử nút n biểu diễn cho một toán tử hai ngôi θ (chẳng hạn + hoặc *), nút con bên trái biểu diễn cho biểu thức E1, nút con bên phải biểu diễn cho biểu thức E2 thì nút n biểu diễn biểu thức E1θ E2, xem hình I.5. Nếu θ là phép toán một ngôi thì nút chứa phép toán θ chỉ có một nút con, nút con này biểu diễn cho toán hạng của θ.
- Khi chúng ta duyệt một cây biểu diễn một biểu thức toán học và liệt kê nhãn của các nút theo thứ tự duyệt thì ta có:
  - Biểu thức dạng tiền tố (prefix) tương ứng với phép duyệt tiền tự của cây.
  - Biểu thức dạng trung tố (infix) tương ứng với phép duyệt trung tự của cây.
  - Biểu thức dạng hậu tố (posfix) tương ứng với phép duyệt hậu tự của cây.

Ví dụ: đối với cây trong hình I.4 ta có:

- Biểu thức tiền tố: *+ab-ac
- Biểu thức trung tố: a+b*a-c
- Biểu thức hậu tố: ab+ac-*

Chú ý

- Các biểu thức này không có dấu ngoặc.
- Các phép toán trong biểu thức toán học có thể có tính giao hoán nhưng khi ta biểu diễn biểu thức trên cây thì phải tuân thủ theo biểu thức đã cho. Ví dụ biểu thức a+b, với a,b là hai số nguyên thì rõ ràng a+b=b+a nhưng hai cây biểu diễn cho hai biểu thức này là khác nhau (vì cây có thứ tự).

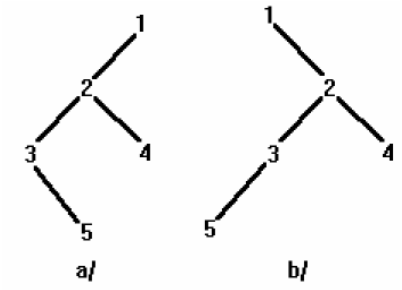
Hình I.6: Cây biểu diễn biểu thức a+b và b+a

- Chỉ có cây ở phía bên trái của hình I.6 mới đúng là cây biểu diễn cho biểu thức a+b theo qui tắc trên.
- Nếu ta gặp một dãy các phép toán có cùng độ ưu tiên thì ta sẽ kết hợp từ trái sang phải. Ví dụ a+b+c-d = ((a+b)+c)-d.

II. Cây nhị phân (Binary Trees)

1. Định nghĩa

Cây nhị phân là cây rỗng hoặc là cây mà mỗi nút có tối đa hai nút con. Hơn nữa các nút con của cây được phân biệt thứ tự rõ ràng, một nút con gọi là nút con trái và một nút con gọi là nút con phải. Ta qui ước vẽ nút con trái bên trái nút cha và nút con phải bên phải nút cha, mỗi nút con được nối với nút cha của nó bởi một đoạn thẳng. Ví dụ các cây trong hình I.7.



Hình I.7: Hai cây có thứ tự giống nhau nhưng là hai cây nhị phân khác nhau

Chú ý rằng, trong cây nhị phân, một nút con chỉ có thể là nút con trái hoặc nút con phải, nên có những cây có thứ tự giống nhau nhưng là hai cây nhị phân khác nhau. Ví dụ hình I.7 cho thấy hai cây có thứ tự giống nhau nhưng là hai cây nhị phân khác nhau. Nút 2 là nút con trái của cây a/ nhưng nó là con phải trong cây b/. Tương tự nút 5 là con phải trong cây a/ nhưng nó là con trái trong cây b/.

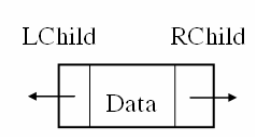
2. Vài tính chất của cây nhị phân

Gọi h và n lần lượt là chiều cao và số phần tử của cây nhị phân. Ta có các tính chất sau:

- Số nút ở mức i<=2^{i+1}. Do đó số nút tối đa của nó là 2^{h-1}
- Số nút tối đa trong cây nhị phân là 2^h-1, hay n<=2^h-1. Do đó chiều cao của nó: n>=h>=$\log_2(n+1)$

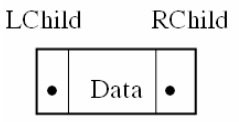
3. Biểu diễn cây nhị phân

Ta chọn cấu trúc động để biểu diễn cây nhị phân:



Trong đó: Lchild, Rchild lần lượt là các con trỏ chỉ đến nút con bên trái và nút con bên phải. Nó sẽ bằng rỗng nếu không có nút con.

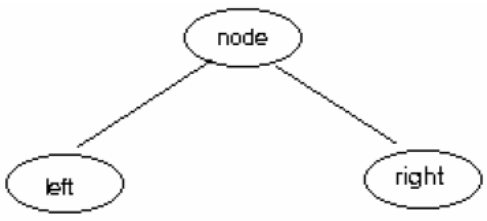
Nút lá có dạng



4. Duyệt cây nhị phân

Ta có thể áp dụng các phép duyệt cây tổng quát để duyệt cây nhị phân. Tuy nhiên vì cây nhị phân là cấu trúc cây đặc biệt nên các phép duyệt cây nhị phân cũng đơn giản hơn. Có ba cách duyệt cây nhị phân thường dùng (xem kết hợp với hình I.8):

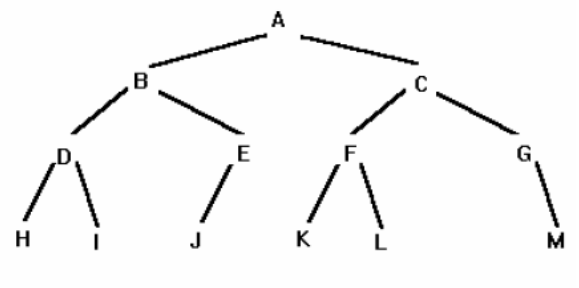
- Duyệt tiền tự (Node-Left-Right): duyệt nút gốc, duyệt tiền tự con trái rồi duyệt tiền tự con phải.
- Duyệt trung tự (Left-Node-Right): duyệt trung tự con trái rồi đến nút gốc sau đó là duyệt trung tự con phải.
- Duyệt hậu tự (Left-Right-Node): duyệt hậu tự con trái rồi duyệt hậu tự con phải sau đó là nút gốc.



HìnhI.8

Chú ý rằng danh sách duyệt tiền tự, hậu tự của cây nhị phân trùng với danh sách duyệt tiền tự, hậu tự của cây đó khi ta áp dụng phép duyệt cây tổng quát. Nhưng danh sách duyệt trung tự thì khác nhau.

Ví dụ



Hình I.9

	Các danh sách duyệt cây nhị phân	Các danh sách duyệt cây tổng quát
Tiền tự:	ABDHIEJCFKLGM	ABDHIEJCFKLGM
Trung tự:	HDIBJEAKFLC**GM**	HDIBJEAKFLC**MG**
Hậu tự:	HIDJEBKLFMGCA	HIDJEBKLFMGCA

5. Cài đặt cây nhị phân

Tương tự cây tổng quát, ta cũng có thể cài đặt cây nhị phân bằng con trỏ bằng cách thiết kế mỗi nút có hai con trỏ, một con trỏ trỏ nút con trái, một con trỏ trỏ nút con phải, trường Data sẽ chứa nhãn của nút.

```
typedef ... TData;
typedef struct Tnode
{
    TData Data;
    TNode* left,right;
};
typedef TNode* TTree;
```

Với cách khai báo như trên ta có thể thiết kế các phép toán cơ bản trên cây nhị phân như sau :

Tạo cây rỗng

Cây rỗng là một cây là không chứa một nút nào cả. Như vậy khi tạo cây rỗng ta chỉ cần cho cây trỏ tới giá trị NULL.

```
void MakeNullTree(TTree *T)
```

```
{
    (*T)=NULL;
}
```

Kiểm tra cây rỗng

```
int EmptyTree(TTree T)
{
    return T==NULL;
}
```

Xác định con trái của một nút

```
TTree LeftChild(TTree n)
{
    if (n!=NULL) return n->left;
    else return NULL;
}
```

Xác định con phải của một nút

```
TTree RightChild(TTree n)
{
    if (n!=NULL) return n->right;
    else return NULL;
}
```

Kiểm tra nút lá:

Nếu nút là nút lá thì nó không có bất kỳ một con nào cả nên khi đó con trái và con phải của nó cùng bằng NULL

```
int IsLeaf(TTree n)
{
    if(n!=NULL)
            return(LeftChild(n)==NULL)&&(RightChild(n)==NULL);
    else return NULL;
}
```

Xác định số nút của cây

```
int nb_nodes(TTree T)
{
```

```
    if(EmptyTree(T)) return 0;
    else return 1+nb_nodes(LeftChild(T))+   nb_nodes(RightChild(T));
}
```

Các thủ tục duyệt cây: tiền tự, trung tự, hậu tự

Thủ tục duyệt tiền tự

```
void PreOrder(TTree T)
{
    cout<<T->Data;
    if (LeftChild(T)!=NULL) PreOrder(LeftChild(T));
    if (RightChild(T)!=NULL)PreOrder(RightChild(T));
}
```

Thủ tục duyệt trung tự

```
void InOrder(TTree T)
{
    if (LeftChild(T)=!NULL)InOrder(LeftChild(T));
    cout<<T->data;
    if (RightChild(T)!=NULL) InOrder(RightChild(T));
}
```

Thủ tục duyệt hậu tự

```
void PosOrder(TTree T)
{
    if (LeftChild(T)!=NULL) PosOrder(LeftChild(T));
    if (RightChild(T)!=NULL)PosOrder(RightChild(T));
    cout<<T->data;
}
```

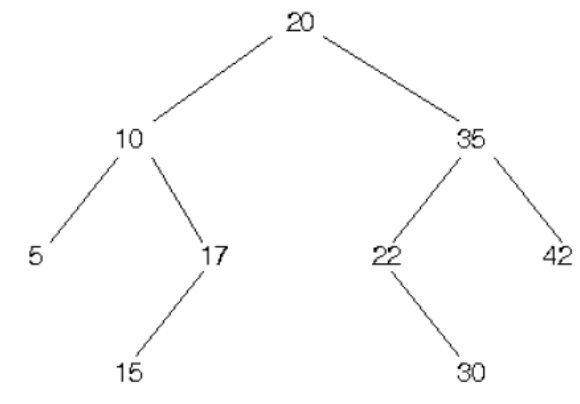
IV. Cây tìm kiếm nhị phân (Binary Search Trees)

1. Định nghĩa

Cây tìm kiếm nhị phân (TKNP) là cây nhị phân mà khoá tại mỗi nút cây lớn hơn khoá của tất cả các nút thuộc cây con bên trái và nhỏ hơn khoá của tất cả các nút thuộc cây con bên phải.

Lưu ý: dữ liệu lưu trữ tại mỗi nút có thể rất phức tạp như là một record chẳng hạn, trong trường hợp này khoá của nút được tính dựa trên một trường nào đó, ta gọi là trường khoá. Trường khoá phải chứa các giá trị có thể so sánh được, tức là nó phải lấy giá trị từ một tập hợp có thứ tự.

Ví dụ: hình I.10 minh hoạ một cây TKNP có khoá là số nguyên (với quan hệ thứ tự trong tập số nguyên).



Hình I.10: Ví dụ cây tìm kiếm nhị phân

Qui ước: Cũng như tất cả các cấu trúc khác, ta coi cây rỗng là cây TKNP

Nhận xét:

- Trên cây TKNP không có hai nút cùng khoá.
- Cây con của một cây TKNP là cây TKNP.
- Khi duyệt trung tự (InOrder) cây TKNP ta được một dãy có thứ tự tăng. Chẳng hạn duyệt trung tự cây trên ta có dãy: 5, 10, 15, 17, 20, 22, 30, 35, 42.

2. Cài đặt cây tìm kiếm nhị phân

Cây TKNP, trước hết, là một cây nhị phân. Do đó ta có thể áp dụng các cách cài đặt như đã trình bày trong phần cây nhị phân. Sẽ không có sự khác biệt nào trong việc cài đặt cấu trúc dữ liệu cho cây TKNP so với cây nhị phân, nhưng tất nhiên, sẽ có sự khác biệt trong các giải thuật thao tác trên cây TKNP như tìm kiếm, thêm hoặc xoá một nút trên cây TKNP để luôn đảm bảo tính chất cuả cây TKNP.

Một cách cài đặt cây TKNP thường gặp là cài đặt bằng con trỏ. Mỗi nút của cây như là một mẩu tin (record) có ba trường: một trường chứa khoá, hai trường kia là hai con trỏ trỏ đến hai nút con (nếu nút con vắng mặt ta gán con trỏ bằng NULL)

Khai báo như sau

```
typedef <kiểu dữ liệu của khoá> KeyType;
typedef struct BSNode
{
    KeyType Key;
    BSNode* Left,Right;
}
typedef BSNode* BSTree;
```

Khởi tạo cây TKNP rỗng

Ta cho con trỏ quản lý nút gốc (Root) của cây bằng NULL.

```
void MakeNullTree(BSTree &Root)
{
    Root=NULL;
}
```

Tìm kiếm một nút có khóa cho trước trên cây TKNP

Để tìm kiếm 1 nút có khoá x trên cây TKNP, ta tiến hành từ nút gốc bằng cách so sánh khoá của nút gốc với khoá x.

- Nếu nút gốc bằng NULL thì không có khoá x trên cây.
- Nếu x bằng khoá của nút gốc thì giải thuật dừng và ta đã tìm được nút chứa khoá x.
- Nếu x lớn hơn khoá của nút gốc thì ta tiến hành (một cách đệ qui) việc tìm khoá x trên cây con bên phải.
- Nếu x nhỏ hơn khoá của nút gốc thì ta tiến hành (một cách đệ qui) việc tìm khoá x trên cây con bên trái.

Ví dụ: tìm nút có khoá 30 trong cây ở trong hình I.10

- So sánh 30 với khoá nút gốc là 20, vì 30 > 20 vậy ta tìm tiếp trên cây con bên phải, tức là cây có nút gốc có khoá là 35.
- So sánh 30 với khoá của nút gốc là 35, vì 30 < 35 vậy ta tìm tiếp trên cây con bên trái, tức là cây có nút gốc có khoá là 22.
- So sánh 30 với khoá của nút gốc là 22, vì 30 > 22 vậy ta tìm tiếp trên cây con bên phải, tức là cây có nút gốc có khoá là 30.
- So sánh 30 với khoá nút gốc là 30, 30 = 30 vậy đến đây giải thuật dừng và ta tìm được nút chứa khoá cần tìm.

Hàm dưới đây trả về kết quả là con trỏ trỏ tới nút chứa khoá x hoặc NULL nếu không tìm thấy khoá x trên cây TKNP.

```
BSTree Search(KeyType x,BSTree Root)
{
    if(Root == NULL)
            return NULL; //không tìm thấy khoá x
    else if (Root->Key == x) /* tìm thấy khoá x */
            return Root;
    else if (Root->Key < x)
    //tìm tiếp trên cây bên phải
            return Search(x,Root->right);
```

```
    else
    {
        tìm tiếp trên cây bên trái
    }
    return Search(x,Root->left);
}
```

Thuật toán tìm kiếm dạng lặp, trả về con trỏ chứa dữ liệu cần tìm và đồng thời giữ lại nút cha của nó nếu tìm thấy, ngược lại trả về rỗng.

```
BSTree SearchLap(BSTree Root, KeyType Item, BSTree &Parent)
{
    BSTree LocPtr = Root;
    Parent = NULL;
    while (LocPtr != NULL)
    {
        if (Item==LocPtr->Key)
            return (LocPtr);
        else
        {
            Parent = LocPtr;
            if (Item > LocPtr->Key)
                LocPtr = LocPtr->RChild;
             else  LocPtr = LocPtr->LChild;
        }
        return(NULL);
    }
}
```

Nhận xét: giải thuật này sẽ rất hiệu quả về mặt thời gian nếu cây TKNP được tổ chức tốt, nghĩa là cây tương đối "cân bằng".

Thêm một nút có khóa cho trước vào cây TKNP

Theo định nghĩa cây tìm kiếm nhị phân ta thấy trên cây tìm kiếm nhị phân không có hai nút có cùng một khoá. Do đó nếu ta muốn thêm một nút có khoá x vào cây TKNP thì trước hết ta phải tìm kiếm để xác định có nút nào chứa khoá x chưa. Nếu có thì giải thuật kết thúc (không làm gì cả!). Ngược lại, sẽ thêm một nút mới chứa khoá x này.

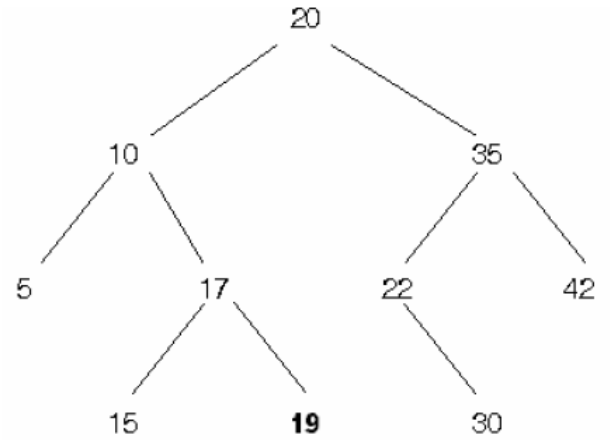
Việc thêm một khoá vào cây TKNP là việc tìm kiếm và thêm một nút, tất nhiên, phải đảm bảo cấu trúc cây TKNP không bị phá vỡ. Giải thuật cụ thể như sau:

Ta tiến hành từ nút gốc bằng cách so sánh khóa cuả nút gốc với khoá x.

- Nếu nút gốc bằng NULL thì khoá x chưa có trên cây, do đó ta thêm một nút mới chứa khoá x.
- Nếu x bằng khoá của nút gốc thì giải thuật dừng, trường hợp này ta không thêm nút.
- Nếu x lớn hơn khoá của nút gốc thì ta tiến hành (một cách đệ qui) giải thuật này trên cây con bên phải.
- Nếu x nhỏ hơn khoá của nút gốc thì ta tiến hành (một cách đệ qui) giải thuật này trên cây con bên trái.

Ví dụ: thêm khoá 19 vào cây ở trong hình I.11

- So sánh 19 với khoá của nút gốc là 20, vì 19 < 20 vậy ta xét tiếp đến cây bên trái, tức là cây có nút gốc có khoá là10.
- So sánh 19 với khoá của nút gốc là 10, vì 19 > 10 vậy ta xét tiếp đến cây bên phải, tức là cây có nút gốc có khoá là 17.
- So sánh 19 với khoá của nút gốc là 17, vì 19 > 17 vậy ta xét tiếp đến cây bên phải. Nút con bên phải bằng NULL, chứng tỏ rằng khoá 19 chưa có trên cây, ta thêm nút mới chứa khoá 19 và nút mới này là con bên phải của nút có khoá là 17, xem hình I.11



Hình I.11: Thêm khoá 19 vào cây hình I.10

Thủ tục sau đây tiến hành việc thêm một khoá vào cây TKNP.

```
void InsertNode(KeyType x, BSTree &Root )
{
    if (Root == NULL)
    { /* thêm nút mới chứa khoá x */
            Root = new BSNode;
```

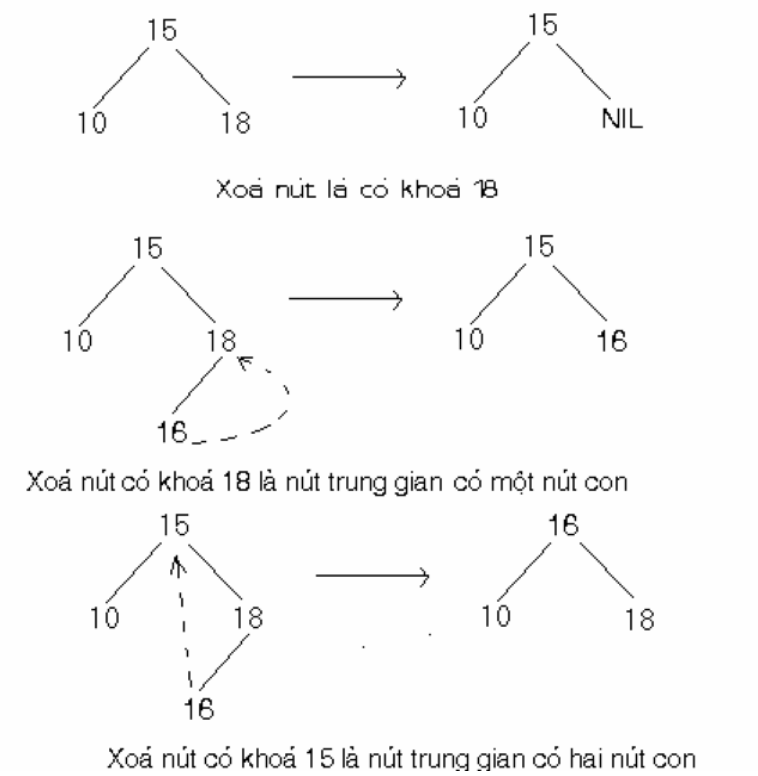
```
            Root->Key = x;
            Root->left = NULL;
            Root->right = NULL;
    }
    else if (x < Root->Key) InsertNode(x,Root->left);
    else if (x>Root->Key)InsertNode(x,Root->right);
}
```

Thủ tục lặp thêm một nút vào cây

```
int InsertNodeLap(BSTree  &Root, KeyType Item)
{
    BSTree LocPtr, Parent;
    if  (SearchLap(Root, Item, Parent))
    {
            cout << "\nđã có ptu  "<<Item<<" trong cây !" ;
            return -1;
    }
    else
    {
            If (LocPtr=CreateNode())==NULL)
                    return 0;
            LocPtr->Key = Item;
            LocPtr->LChild = NULL;
            LocPtr->RChild = NULL;
            if  (Parent == NULL)
                    Root = LocPtr;   // cây rỗng
             else    if (Item < Parent->Data)
                    Parent->LChild =  LocPtr;
            else Parent->RChild = LocPtr;
            return 1;
     }
}
```

Xóa một nút có khóa cho trước ra khỏi cây TKNP

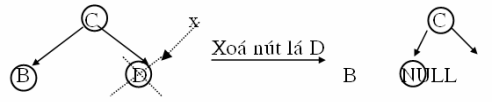
Giả sử ta muốn xoá một nút có khoá x, trước hết ta phải tìm kiếm nút chứa khoá x trên cây.



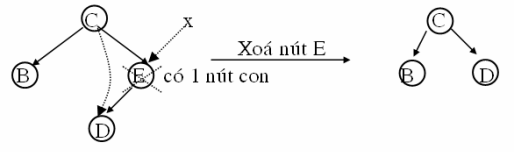
Hình I.12

Việc xoá một nút như vậy, tất nhiên, ta phải bảo đảm cấu trúc cây TKNP không bị phá vỡ.

- Nếu không tìm thấy nút chứa khoá x thì giải thuật kết thúc.
- Nếu tìm gặp nút N có chứa khoá x, ta có ba trường hợp sau
  - Nếu N là lá ta thay nó bởi NULL.

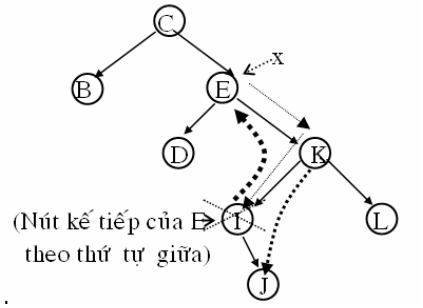


  - N chỉ có một nút con ta thay nó bởi nút con của nó.



  - N có hai nút con ta thay nó bởi nút lớn nhất trên cây con trái của nó (nút cực phải của cây con trái) hoặc là nút bé nhất trên cây con phải của nó (nút cực trái của cây con phải). Trong giải thuật sau, ta thay x bởi khoá của nút cực trái của cây con bên phải rồi ta xoá nút cực trái này. Việc

xoá nút cực trái của cây con bên phải sẽ rơi vào một trong hai trường hợp trên.



Hình I.12

Giải thuật xoá một nút có khoá nhỏ nhất

Hàm dưới đây trả về khoá của nút cực trái, đồng thời xoá nút này.

```
KeyType DeleteMin (BSTree &Root )
{
    KeyType k;
    if (Root->left == NULL)
    {
            k=Root->key;
            Root = Root->right;
            return k;
    }
    else return DeleteMin(Root->left);
}
```

Thủ tục xóa một nút có khoá cho trước trên cây TKNP

```
void DeleteNode(KeyType  x, BSTree &Root)
{
    if (Root != NULL)
    if (x < Root->Key) DeleteNode(x,Root->left)
    else if (x  > Root->Key)
            DeleteNode(x,Root->right)
    else if ((Root->left==NULL) && (Root->right==NULL))
            Root =NULL;
    else if (Root->left == NULL)
            Root = Root->right
```

```
    else if (Root->right==NULL)
            Root = Root->left
    else Root->Key = DeleteMin(Root->righ)
}
```

Thủ tục lặp xóa một node ra khỏi cây

```
int DeleteNode (BSTree  &Root, KeyType Item)
{
    BSTree x, Parent, xSucc, SubTree;
    if((x=SearchLap(Root,Item,Parent)) == NULL)
            return 0;   //không thấy Item
    else
    {
            if((x->left!=NULL)&&(x->right != NULL))
            // nút có hai con
            {
                    xSucc =  x->right;
                    Parent = x;
                    while (xSucc->left != NULL)
                    {
                            Parent = xSucc;
                            xSucc =  xSucc->left;
                    }
                    x->Key = xSucc->Key;
                    x = xSucc;
            }
            //đã đưa nút 2 con về nút có tối đa 1 con
            SubTree =  x->left;
            if  (SubTree ==  NULL)
                    SubTree =  x->right;
            if  (Parent == NULL)
                    Root =  SubTree; // xóa nút gốc
            else if (Parent->left == x)
```

```
                Parent->left = SubTree;
            else  Parent->right = SubTree;
            delete  x;
        return 1;
    }
}
```

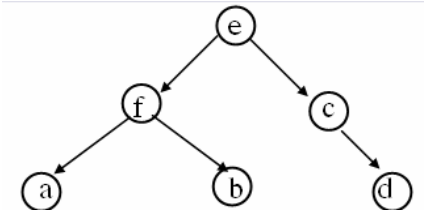
V. Cây nhị phân tìm kiếm cân bằng (Cây AVL)

1. Cây nhị phân cân bằng hoàn toàn

Định nghĩa

Cây nhị phân cân bằng hoàn toàn (CBHT) là cây nhị phân mà đối với mỗi nút của nó, số nút của cây con trái chênh lệch không quá 1 so với số nút của cây con phải.

Ví dụ:



Hình I.13

2. Xây dựng cây nhị phân cân bằng hoàn toàn

```
Tree  CreateTreeCBHT(int  n)
{
        Tree Root;
        int nl, nr;
        KeyType x;
        if (n<=0) return NULL;
        nl = n/2; nr = n-nl-1;
        Input(x);//nhập phần tử x
        if ((Root =CreateNode()) == NULL)
                return NULL;
        Root->Key = x;
        Root->left = CreateTreeCBHT(nl);
        Root->right = CreateTreeCBHT(nr);
        return Root;
}
```

3. Cây tìm kiếm nhị phân cân bằng (cây AVL)

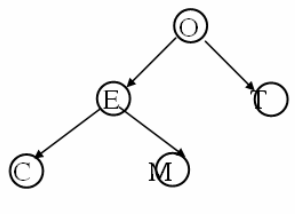
Trên cây nhị phân tìm kiếm BST có n phần tử mà là cây CBHT, phép tìm kiếm một phần tử trên nó sẽ thực hiện rất nhanh: trong trường hợp xấu nhất, ta chỉ cần thực hiên $\log_2 n$ phép so sánh. Nhưng cây CBHT có cấu trúc kém ổn định trong các thao tác cập nhật cây, nên nó ít được sử dụng trong thực tế. Vì thế, người ta tận dụng ý tưởng cây cân bằng hoàn toàn để xây dựng một cây nhị phân tìm kiếm có trạng thái cân bằng yếu hơn, nhưng việc cân bằng lại chỉ xảy ra ở phạm vi cục bộ đồng thời chi phí cho việc tìm kiếm vẫn đạt ở mức $O(\log_2 n)$. Đó là cây tìm kiếm cân bằng.

Định nghĩa

Cây nhị phân tìm kiếm gọi là cây nhị phân tìm kiếm cân bằng (gọi tắt là cây AVL) nếu tại mỗi nút của nó, độ cao của cây con trái và độ cao của cây con phải chênh lệch nhau không quá 1.

Rõ ràng một cây nhị phân tìm kiếm cân bằng hoàn toàn là cây cân bằng, nhưng điều ngược lại là không đúng. Chẳng hạn cây nhị phân tìm kiếm trong ví dụ sau là cân bằng nhưng không phải là cân bằng hoàn toàn.

Ví dụ:



Hình I.14

Cây cân bằng AVL vẫn thực hiện việc tìm kiếm nhanh tương đương cây cân bằng hoàn toàn và vẫn có cấu trúc ổn định hơn hẳn cây cân bằng.

Chỉ số cân bằng và việc cân bằng lại cây AVL

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

Kí hiệu:

$h_l(p)$ hay h_l là chiều cao của cây con trái của p.

$h_r(p)$ hay h_r là chiều cao của cây con phải của p.

EH = 0, RH = 1, LH = -1

CSCB(p) = EH	⇔	$h_R(p) = h_L(p)$:2 cây con cao bằng nhau
CSCB(p) = RH	⇔	$h_R(p) > h_L(p)$: cây lệch phải
CSCB(p) = LH	⇔	$h_R(p) < h_L(p)$: cây lệch trái

Với mỗi nút của cây AVL, ngoài các thuộc tính thông thường như cây nhị phân, ta cần lưu ý thêm thông tin về chỉ số cân bằng trong cấu trúc của một nút. Ta có định nghĩa cấu trúc một nút như sau:

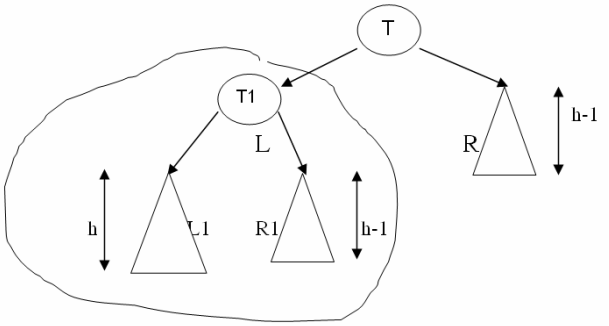
```
typedef    .....   ElementType;  /* Kiểu dữ liệu của nút */
typedef  struct  AVLTN
{
    ElementType Data;
    int  Balfactor; //Chỉ số cân bằng
    struct AVLTN * Lchild, *Rchild;
} AVLTreeNode;
typedef   AVLTreeNode  *AVLTree;
```

Việc thêm hay hủy một nút trên cây AVL có thể làm cây tăng hay giảm chiều cao, khi đó ta cần phải cân bằng lại cây. Để giảm tối đa chi phí cân bằng lại cây, ta chỉ cân bằng lại cây AVL ở phạm vi cục bộ.

Các trường hợp mất cân bằng

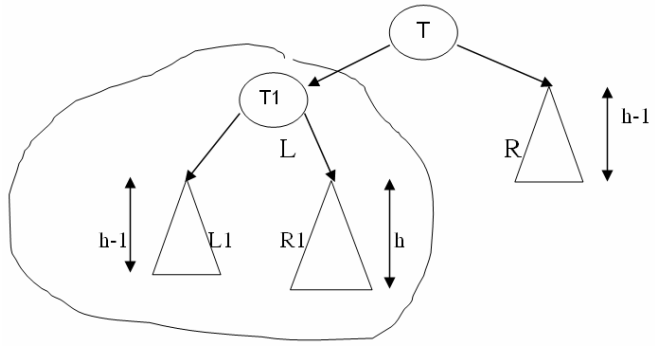
Ngoài các thao tác thêm và hủy đối với cây cân bằng, ta còn có thêm thao tác cơ bản là cân bằng lại cây AVL trong trường hợp thêm hoặc hủy một nút của nó. Khi đó độ lệch giữa chiều cao cây con phải và trái sẽ là 2. Do đó trường hợp cây lệch trái và phải tương ứng là đối xứng nhau, nên ta chỉ xét trường hợp cây AVL lệch trái.

Trường hợp a: cây con T1 lệch trái



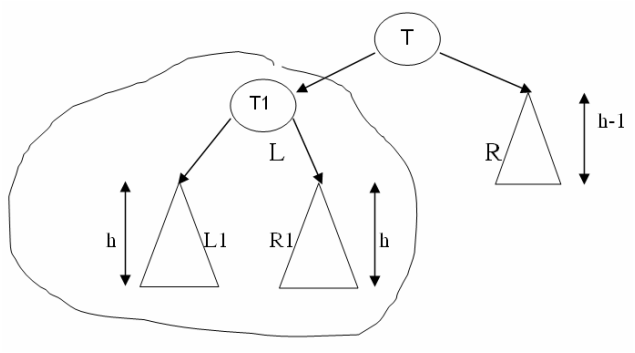
Hình I.15

Trường hợp b: cây con T1 lệch phải



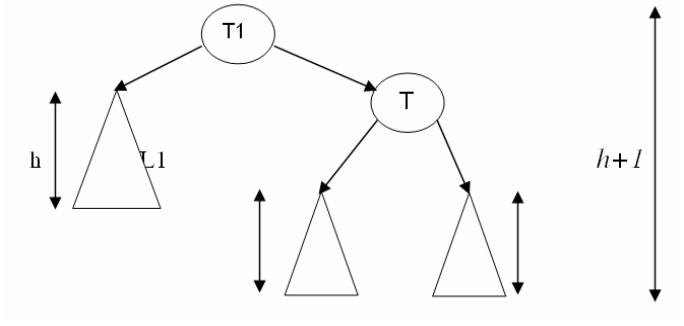
Hình I.16

Trường hợp c: cây con T1 không lệch



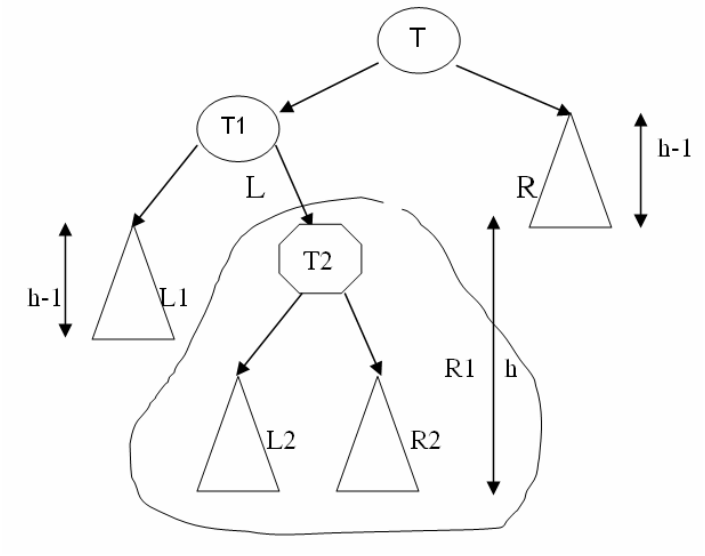
Hình I.17

Cân bằng lại trường hợp a: ta cân bằng lại bằng phép quay đơn left-left ta được:



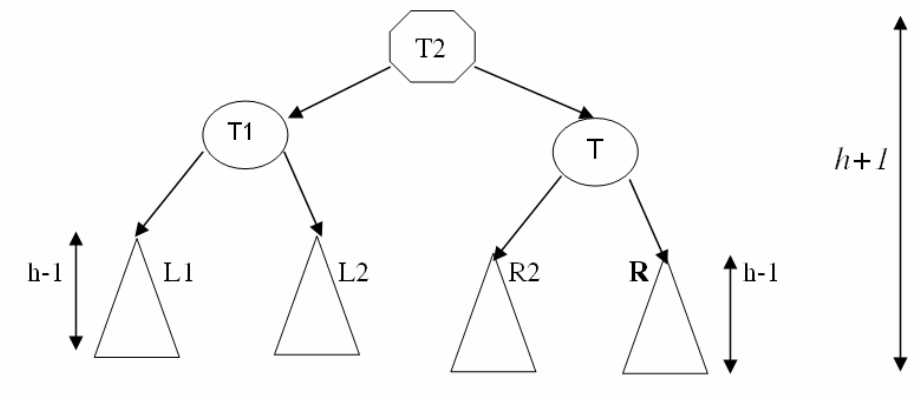
Hình I.18

Cân bằng lại trường hợp b:



Hình I.19

Cân bằng lại bằng phép quay kép left-right, ta có kết quả như sau:



Hình I.20

Cài đặt

```
//Phép quay đơn Left – Left
void RotateLL(AVLTree &T)
{
    AVLTree T1 = T->Lchild;
    T->Lchild = T1->Rchild;
    T1->Rchild = T;
    switch (T1->Balfactor)
    {
            case LH: T->Balfactor = EH;
                T1->Balfactor = EH; break;
            case EH: T->Balfactor = LH;
                T1->Balfactor = RH; break;
    }
    T = T1;
    return ;
}
// Phép quay đơn Right – Right
void RotateRR (AVLTree &T)
{
    AVLTree T1 = T->Rchild;
    T->Rchild = T1->Lchild;
    T1->Lchild = T;
    switch (T1->Balfactor)
```

```
    {
            case RH: T->Balfactor = EH;
                T1->Balfactor = EH; break;
            case EH: T->Balfactor = RH;
                T1->Balfactor = LH; break;
    }
    T = T1;
    return ;
}
//Phép quay kép Left – Right
void RotateLR(AVLTree &T)
{
    AVLTree T1 = T->Lchild, T2 = T1->Rchild;
    T->Lchild = T2->Rchild; T2->Rchild = T;
    T1->Rchild = T2->Lchild; T2->Lchild = T1;
    switch (T2->Balfactor)
    {
            case LH: T->Balfactor = RH;
                T1->Balfactor = EH; break;
            case EH: T->Balfactor = EH;
                T1->Balfactor = EH; break;
            case RH: T->Balfactor = EH;
                T1->Balfactor = LH; break;
    }
    T2->Balfactor = EH;
    T = T2;
    return ;
}
//Phép quay kép Right-Left
void RotateRL(AVLTree &T)
{
    AVLTree T1 = T->RLchild, T2 = T1->Lchild;
```

```
    T->Rchild = T2->Lchild; T2->Lchild = T;
    T1->Lchild = T2->Rchild; T2->Rchild = T1;
    switch (T2->Balfactor)
    {
            case LH: T->Balfactor = EH;
                T1->Balfactor = RH; break;
            case EH: T->Balfactor = EH;
                T1->Balfactor = EH; break;
            case RH: T->Balfactor = LH;
                T1->Balfactor = EH; break;
    }
    T2->Balfactor = EH;
    T = T2;
    return ;
}
```

Cài đặt các thao tác cân bằng lại

```
//Cân bằng lại khi cây bị lệch trái
int LeftBalance(AVLTree &T)
{
    AVLTree T1 = T->Lchild;
    switch (T1->Balfactor)
    {
            case LH : RotateLL(T);
                    return 2; //cây T không bị lệch
            case EH : RotateLL(T);
                    return 1;//cây T bị lệch phải
    case RH : RotateLR(T); return 2;
    }
    return  0;
}
//Cân bằng lại khi cây bị lệch phải
int RightBalance(AVLTree &T)
```

```
{
    AVLTree T1 = T->Rchild;
    switch (T1->Balfactor)
    {
            case LH : RotateRL(T);
                    return 2; //cây T không lệch
            case EH : RotateRR(T);
                    return 1; //cây T lệch trái
    case RH : RotateRR(T); return 2;
    }
    return  0;
}
```

Chèn một phần tử vào cây AVL

Việc chèn một phần tử vào cây AVL xảy ra tương tự như trên cây nhị phân tìm kiếm. Tuy nhiên sau khi chèn xong, nếu chiều cao của cây thay đổi tại vị trí thêm vào, ta cần phải ngược lên gốc để kiểm tra xem có nút nào bị mất cân bằng hay không. Nếu có, ta chỉ cần phải cân bằng lại ở nút này.

```
AVLTree CreateAVL()
{
    AVLTree Tam= new AVLTreeNode;
    if (Tam == NULL)
            cout << "\nLỗi !";
    return Tam;
}
int  InsertNodeAVL( AVLTree &T, ElementType x)
{
    int  Kqua;
    if (T)
    {
            if(T->Data==x)
                    return 0; //Đã có nút trên cây
            if (T-> Data >  x)
            {
```

```
        //chèn nút vào cây con trái
        Kqua = InsertNodeAVL(T->Lchild,x);
        if (Kqua < 2) return  Kqua;
        switch  (T->Balfactor)
        {
                case LH: LeftBalance(T);
                        return 1;//T lệch trái
                case EH: T->Balfactor=LH;
                        return 2;//T không lệch
                caseRH:T->Balfactor=EH;
                        return 1;//T lệch phải
        }
}
else   // T-> Data <  x
{
        Kqua= InsertNodeAVL(T->Rchild,x);
        if (Kqua < 2) return  Kqua;
        switch  (T->Balfactor)
        {
                case LH: T->Balfactor = EH;
                        return 1;
                case EH:T->Balfactor=RH;
                        return 2;
                case RH : RightBalance(T);
                        return 1;
        }
}
else //T==NULL
{
        if ((T = CreateAVL()) == NULL)
                return –1;
        T->Data = x;
```

```
            T->Balfactor = EH;
            T->Lchild  = T->Rchild = NULL;
            return 2;
        }
    }
```

Xóa một phần tử ra khỏi cây AVL

Việc xóa một phần tử ra khỏi cây AVL diễn ra tương tự như đối với cây nhị phân tìm kiếm, chỉ khác là sau khi hủy, nếu cây AVL bị mất cân bằng, ta phải cân bằng lại cây. Việc cân bằng lại cây có thể xảy ra phản ứng dây chuyền.

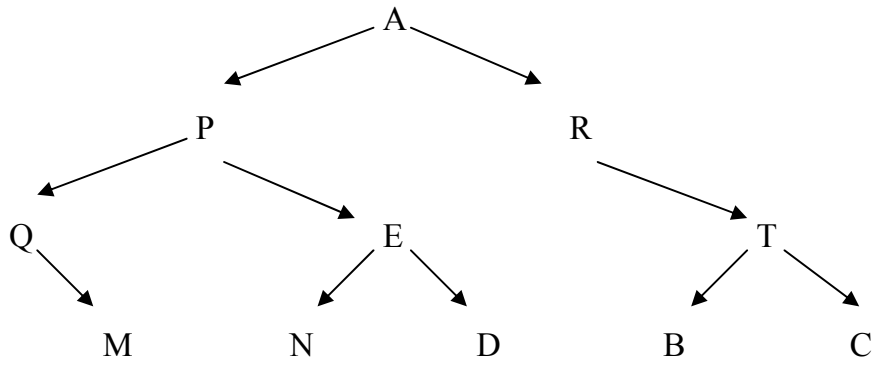
```
int DeleteAVL(AVLTree &T, ElementType x)
{
    int Kqua;
    if (T== NULL) return 0; // không có x trên cây
    if  (T-> Data >  x)
    {
        Kqua = DeleteAVL(T->Lchild,x);// tìm và xóa x trên cây con trái của T
        if (Kqua < 2) return  Kqua;
        switch  (T->Balfactor)
        {
            case LH : T->Balfactor = EH;
                return 2; //trước khi xóa T lệch trái
            case EH : T->Balfactor = RH;
                return 1;//trước khi xóa T không lệch
        case RH : return RightBalance(T);
            // trước khi xóa T lệch phải
        }
    }
    else if (T-> Data <  x)
    {
        Kqua = DeleteAVL(T->Rchild,x);// tìm và xóa x trên cây con trái của T
        if (Kqua < 2) return  Kqua;
        switch  (T->Balfactor)
```

```
        {
                case LH : return LeftBalance(T);//trước khi xóa T lệch trái
                case EH : T->Balfactor = LH;
                        return 1;       //trước khi xóa T không lệch
                 case RH : T->Balfactor = EH;
                return 2; //trước khi xóa T lệch phải
        }
}
else //T->Data== x
{
        AVLTree p = T;
        if (T->Lchild == NULL)
        {
                T = T->Rchild; Kqua = 2;
        }
        else    if  (T->Rchild == NULL)
        {
                T = T->Lchild; Kqua = 2;
        }
        else // T có hai con
        {
                Kqua = TimPhanTuThayThe(p,T->Rchild);
                //Tìm phần tử thay thế P để xóa trên nhánh phải cuả T
                if (Kqua < 2) return Kqua;
                switch (T->Balfactor)
                {
                        case  LH : return LeftBalnce(T);
                        case  EH : T->Balfactor = LH;
                                return 2;
                        case  RH : T->Balfactor = EH;
                                return 2;
                }
        }
```

```
            delete p;
            return Kqua;
        }
    }
    // Tìm phần tử thay thế
    int TimPhanTuThayThe(AVLTree  &p, AVLTree  &q)
    {
        int Kqua;
        if (q->Lchild)
        {
            Kqua = TimPhanTuThayThe(p, q->Lchild);
            if (Kqua < 2) return Kqua;
            switch (q->Balfactor)
            {
                case  LH : q->Balfactor = EH;
                        return 2;
                case  EH : q->Balfactor = RH;
                        return 1;
                case  RH : return RightBalance(q);
            }
        }
        Else
        {
            p->Data = q->Data;
            p = q;
            q = q->Rchild;
            return 2;
        }
    }
```

Bài tập

1. Xuất ra theo thứ tự: giữa, đầu, cuối các phần tử trên cây nhị phân sau:



2. Tìm cây nhị phân thỏa đồng thời hai điều kiện kết xuất sau:
   - Theo thứ tự đầu NLR của nó là dãy ký tự sau:

     A, B, C, D, E, Z, U, T, Y
   - Theo thứ tự giữa LNR của nó là dãy ký tự sau:

     D, C, E, B, A, U, Z, T, Y
3. Biểu diễn mỗi biểu thứ số học dưới đây trên cây nhị phân, từ đó rút ra dạng biểu thức hậu tố của chúng:
   - a/(b*c)
   - $a^5 + 4a^3 – 3a^2 + 7$
   - (a + b) * (c - d)
   - S^{a+b}

   Viết thuật toán vàchương trình:
   - Chuyển một biểu thức số học ký hiệu lên cây nhị phân (có kiểm tra biểu thức đã cho có hợp cú pháp không ?)
   - Xuất ra biểu thức số học đó dưới dạng: trung tố, hậu tố, tiền tố
4. Xây dựng cây tìm kiếm nhị phân BST từ mỗi bộ mục dữ liệu đầu vào như sau:
   - 1,2,3,4,5
   - 5,4,3,2,1
   - fe, cx, jk, ha, ap, aa, by, my, da
   - 8,9,11,15,19,20,21,7,3,2,1,5,6,4,13,10,12,17,16,18. Sau đó xóa lần lượt các nút sau: 2,10,19,8,20
5. Viết chương trình với các chức năng sau:
   - Nhập từ bàn phím các số nguyên vào một cây nhị phân tìm kiếm (BST) mà nút gốc được trỏ tới bởi con trỏ Root.
   - Xuất các phần tử trên cây BST trên theo thứ tự: đầu, giữa, cuối.

- Tìm và xóa (nếu có thể) phần tử trên cây Root có dữ liệu trùng với một mục dữ liệu Item cho trước được nhập từ bàn phím.
- Sắp xếp n mục dữ liệu (được cài đặt bằng DSLK) bằng phương pháp cây nhị phân tìm kiếm BSTSort.

Yêu cầu: viết các thao tác trên bằng 2 phương pháp đệ quy và lặp

6. Tương tự bài 5 nhưng trong mỗi nút có thêm trường parent để trỏ tới cha của nó.
7. Cho cây nhị phân T. Viết chương trình chứa các hàm cótácdụng xác định:
   - Tổng số nút của cây.
   - Số nút của cây ở mức k.
   - Số nút lá.
   - Chiều cao của cây.
   - Kiểm tra xem cây T có phải cây cân bằng hoàn toàn hay không?
   - Số nút có đúng hai con khác rỗng
   - Số nút có đúng một con khác rỗng
   - Số nút có khóa nhỏ hơn x trên cây nhị phân hoặc cây BST
   - Số nút có khóa lớn hơn x trên cây nhị phân hoặc cây BST
   - Duyệt theo chiều rộng
   - Duyệt theo chiều sâu
   - Đảo nhánh trái và phải của một cây nhị phân.
8. Viết chương trình thực hiện các thao tác cơ bản trên cây AVL: chèn một nút, xóa một nút, tạo cây AVL, hủy cây AVL.
9. Viết chương trình cho phép tạo, thêm, bớt, tra cứa, sửa chữa từ điển.

Chương II

Đồ Thị

Mục tiêu

Sau khi học xong chương này, sinh viên nắm vững và cài đặt được các kiểu dữ liệu trừu tượng đồ thị và vận dụng để giải những bài toán thực tế.

Kiến thức cơ bản cần thiết

Để học tốt chương này sinh viên cần phải nắm vững kỹ năng lập trình cơ bản như:

- Kiểu mẩu tin, kiểu mảng, kiểu con trỏ.
- Các cấu trúc điều khiển, lệnh vòng lặp.
- Lập trình hàm, thủ tục, cách gọi hàm.

Nội dung

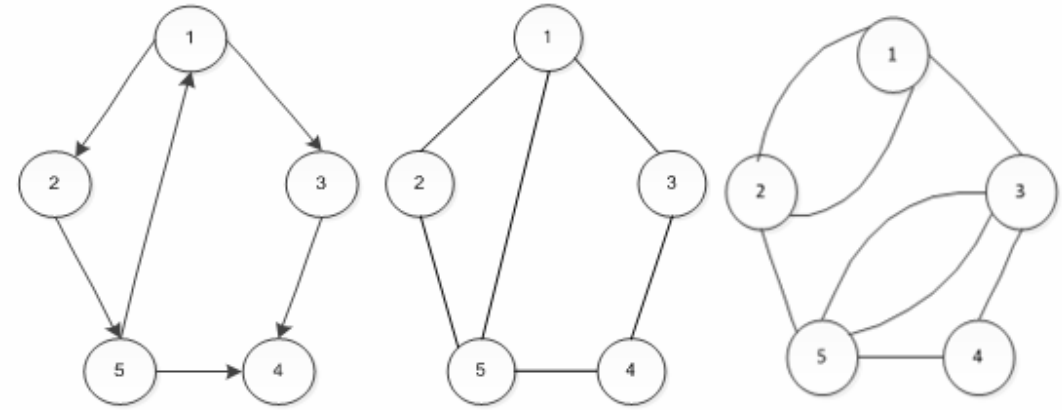
Trong chương này chúng ta sẽ nghiên cứu một số kiểu dữ liệu trừu tượng cơ bản như sau:

- Các khái niệm cơ bản
- Kiểu dữ liệu trừu tượng đồ thị
- Biểu diễn đồ thị
- Các phép duyệt đồ thị
- Một số bài toán trên đồ thị

I. Các định nghĩa

Một đồ thị G = (V, E) là một tập hợp không rỗng V chứa các đỉnh và một tập hợp không rỗng E chứa các cạnh (cung) tương ứng. Các đỉnh còn được gọi là nút hay điểm. Các cung nối giữa hai đỉnh, hai đỉnh này có thể trùng nhau. Số đỉnh và cung kí hiệu tương ứng là |V| và |E|.

Hai đỉnh có cung nối nhau gọi là hai đỉnh kề. Một cung nối giữa hai đỉnh v, w có thể coi như là một cặp điểm (v, w). Nếu cặp này có thứ tự thì ta có cung có thứ tự, ngược lại thì là cung không có thứ tự. Nếu các cung trong đồ thị G có thứ tự (tức cung (v, w) khác cung (w, v)) thì G gọi là đồ thị có hướng. Nếu các cung trong đồ thị G không có thứ tự (tức cung (v, w) = (w, v)) thì đồ thị G gọi là đồ thị vô hướng. Một đồ thị được gọi là đa đồ thị nếu giữa hai đỉnh có thể nối với nhau bởi nhiều hơn một cung, ngược lại thì đồ thị là đơn đồ thị. Hình I.1a: đồ thị có hướng, hình I.1.b: đồ thị vô hướng, hình I.1.c: đa đồ thị. Trong các đồ thị này thì các vòng tròn được đánh số biểu diễn các đỉnh, còn các cung được biểu diễn bằng đoạn nối hai đỉnh có hướng (trong I.1a) hoặc không có hướng (trong I.1b).

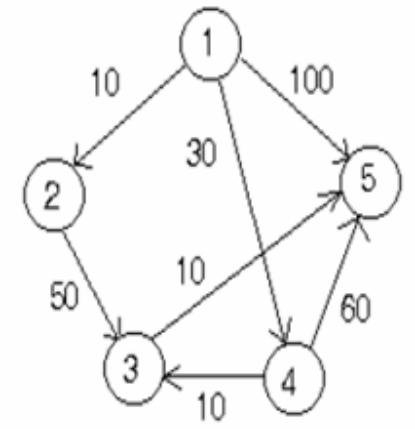


Hình I.1a **Hình I.1.b** **Hình I.1c**

Một đường đi trên đồ thị là một dãy tuần tự các đỉnh v_1, v_2, ..v_n sao cho (v_i, v_{i+1}) là một cung trên đồ thị (i=1,…,n-1). Đường đi này là đường đi từ v_1 đến v_n và đi qua các đỉnh v_2, .., v_{n-1}. Đỉnh v_1 gọi là đỉnh đầu, v_n còn gọi là đỉnh cuối, độ dài đường đi này bằng (n-1). Trường hợp đặc biệt dãy chỉ có một đỉnh v thì ta coi đó là đường đi từ nó đến chính nó và độ dài bằng 0. Ví dụ dãy 1, 2, 5 trong đồ thị I.1.a là một đường đi từ đỉnh 1 đến đỉnh 5, đường đi này có độ dài bằng 2.

Đường đi gọi là đường đi đơn nếu mọi đỉnh trên đường đi đều khác nhau, ngoại trừ đỉnh đầu và đỉnh cuối có thể trùng nhau. Một đường đi có đỉnh đầu và đỉnh cuối trùng nhau gọi là một chu trình. Một chu trình đơn là một đường đi đơn có đỉnh đầu và đỉnh cuối trùng nhau và có độ dài ít nhất là 1. Ví dụ trong hình I.1a thì 3,2,4,3 tạo thành một chu trình có độ dài 3. Trong hình I.1b thì 1,2,5,1 là một chu trình có độ dài bằng 3.

Trong nhiều ứng dụng ta thường kết hợp các giá trị hay nhãn với các đỉnh hoặc các cạnh, lúc này ta có đồ thị có nhãn. Nhãn kết hợp với các đỉnh hoặc cạnh có thể biểu diễn tên, giá, khoảng cách …Nói chung nhãn có thể có kiểu tuỳ ý. Hình I.2 là một đồ thị có nhãn.



Hình I.2

Đồ thị con của một đồ thị G = (V, E) là một đồ thị G’ = (V’, E’) trong đó:

- V’$\subseteq$ V và

- E’ gồm các cạnh (v, w) $\in$ E sao cho v, w $\in$ V’

II. Biểu diễn đồ thị

Thông thường để biểu diễn đồ thị người ta dùng hai cấu trúc dữ liệu là ma trận (ma trận kề) hoặc mảng các danh sách liên kết các đỉnh kề (danh sách kề).

1. Biểu diễn đồ thị bằng ma trận kề

Ta dùng một mảng hai chiều, chẳng hạn mảng DT, kiểu boolean để biểu diễn các đỉnh kề. Nếu đồ thị có n đỉnh thì ta dùng mảng DT kích thước n x n. Giả sử các đỉnh được đánh số 1..n thì DT[i,j] = true, nếu có cạnh nối giữa hai đỉnh i và j, ngược lại DT[i,j] = false. Nếu đồ thị G là đồ thị vô hướng thì ma trận kề sẽ là ma trận đối xứng. Chẳng hạn đồ thị I.1b có biểu diễn ma trận kề như sau:

j / i	**1**	**2**	**3**	**4**	**5**
1	True	True	True	**False**	True
2	True	True	**False**	**False**	True
3	True	**False**	True	True	**False**
4	**False**	**False**	True	True	True
5	True	True	**False**	True	True

Ở đây ta cũng có thể biểu diễn dùng hai giá trị 0 và 1 để biểu diễn, quy ước 1 tương ứng với true còn 0 tương ứng với false. Với cách biểu diễn này thì đồ thị hình I.1a có biểu diễn ma trận kề như sau:

j / i	1	2	3	4	5
1	1	1	1	0	0
2	0	1	0	0	1
3	0	0	1	1	0
4	0	0	0	1	0
5	1	0	0	1	1

Trên đồ thị có nhãn thì ma trận kề có thể dùng để lưu trữ nhãn của các cung chẳng hạn cung giữa i và j có nhãn a thì DT[i,j] = a. Ví dụ ma trận kề của đồ thị hình I.2 là:

j / i	1	2	3	4	5
1	0	10	VC	30	100
2	VC	0	50	VC	VC
3	VC	VC	0	VC	10
4	VC	VC	10	0	60
5	VC	VC	VC	VC	0

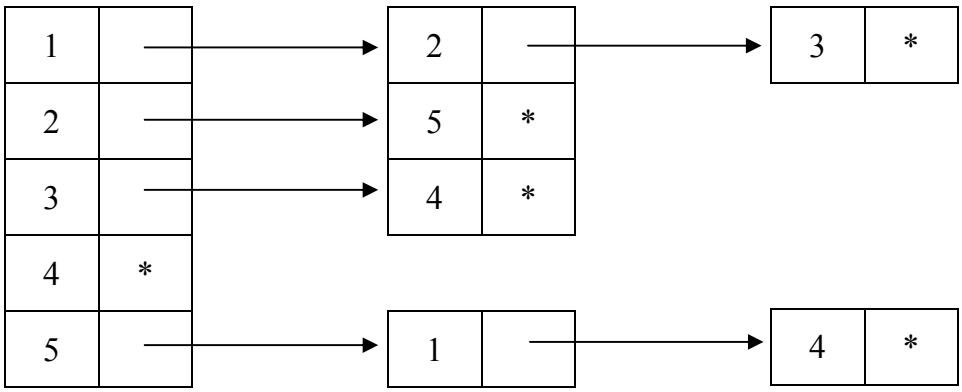
Đối với những cặp đỉnh i, j không có cung nối với nhau ta phải gán cho nó một giá trị đặc biệt nào đó để phân biệt với các giá trị có nghĩa khác. Chẳng hạn như trong bài toán tìm đường đi ngắn nhất, các giá trị số nguyên biểu diễn cho khoảng cách giữa hai thành phố không có cạnh nối ta gán cho nó khoảng cách bằng giá tri VC là một giá trị vô cùng lớn, còn khoảng cách từ một đỉnh đến chính nó là 0.

Bài tập: Hãy viết thủ tục nhập liệu một ma trận kề biểu diễn cho một đồ thị. Dữ liệu đầu vào là số đỉnh V, số cạnh E và các cạnh nối hai đỉnh.

Cách biểu diễn đồ thị bằng ma trận kề cho phép kiểm tra một cách trực tiếp hai đỉnh nào đó có thể kề nhau không. Nhưng nó phải mất thời gian duyệt qua toàn bộ mảng để xác định tất cả các cạnh trên đồ thị. Thời gian này độc lập với số cạnh và số đỉnh của đồ thị. Ngay cả khi số cạnh của đồ thị rất nhỏ thì ta vẫn phải dùng một ma trận nxn để lưu trữ. Do vậy, nếu ta cần làm việc thường xuyên với các cạnh của đồ thị thì ta có thể phải dùng cách biểu diễn khác cho phù hợp hơn.

2. Biểu diễn đồ thị bằng danh sách các đỉnh kề.

Trong cách biểu diễn này, ta sẽ lưu trữ các đỉnh kề với một đỉnh i trong một danh sách liên kết theo một thứ tự nào đó. Như vậy ta cần một mảng LIST một chiều có n phần tử để biểu diễn cho đồ thị có n đỉnh. LIST[i] là con trỏ trỏ tới danh sách các đỉnh kề với đỉnh i. Ví dụ đồ thị hình I.1a có thể biểu diễn như sau:



Mảng LIST

Bài tập: viết thủ tục nhập dữ liệu cho đồ thị biểu diễn bằng danh sách kề.

IV. Các phép duyệt đồ thị (traversals of Graph)

Trong khi giải nhiều bài toán được mô hình hóa bằng đồ thị, ta cần đi qua các đỉnh và các cung của đồ thị một cách có hệ thống. Việc đi qua các đỉnh của đồ thị một cách có hệ thống như vậy gọi là duyệt đồ thị. Có hai phép duyệt đồ thị phổ biến đó là duyệt theo chiều sâu, và duyệt theo chiều rộng.

1. Duyệt theo chiều sâu (Depth-first search)

Giả sử ta có đồ thị G = (V, E) với các đỉnh ban đầu được đánh dấu là chưa duyệt (mảng đánh dấu mang giá trị 0). Từ một đỉnh v nào đó ta bắt đầu duyệt như sau: đánh dấu v đã duyệt, với mỗi đỉnh w chưa duyệt kề với v, ta thực hiện đệ qui quá trình trên cho w. Sở dĩ cách duyệt này có tên là duyệt theo chiều sâu vì nó sẽ duyệt theo một hướng nào đó sâu nhất có thể được. Giải thuật duyệt theo chiều sâu một đồ thị có thể được trình bày như sau, trong đó ta dùng một mảng DX có n phần tử để đánh dấu các đỉnh của đồ thị là đã duyệt hay chưa.

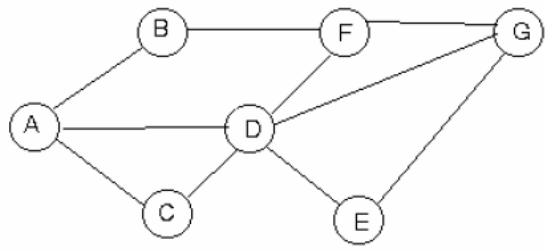
```
//đánh dấu chưa duyệt tất cả các đỉnh
for (v =1; v <= n; v++) DX[v] = 0;
```

Thủ tục duyệt đồ thị theo chiều sâu *dfs* có thể được viết dạng đệ quy như sau:

```
void dfs(đỉnh v) // v thuộc [0..n]
{
    vertex w;
    DX[v]=1;
    for (mỗi đỉnh w là đỉnh kề với v)
            if (DX[w] == 0)
                    dfs(w);
}
```

Ví dụ: duyệt theo chiều sâu đồ thị trong hình I.1.a. Giả sử ta bắt đầu duyệt từ đỉnh 1, tức là dfs(1). Giải thuật sẽ đánh dấu 1 đã được duyệt, 1 có hai đỉnh kề là 2 và 3, chọn đỉnh đầu tiên trong danh sách các đỉnh kề với 1, đó là 2. Tiếp tục duyệt 2, 2 được đánh dấu đã xét, 2 có một đỉnh kề là 5, 5 được đánh dấu đã duyệt, 5 có hai đỉnh kề là 1 và 4, nhưng 1 đã được đánh dấu là xét rồi do đó thuật toán thực hiện duyệt tới đỉnh 4 là dfs(4). Đến đây, không còn đỉnh nào kề với 4, bây giờ giải thuật sẽ tiếp tục với đỉnh kề với 1 mà còn chưa duyệt là 3. Đỉnh 3 không có đỉnh kề nên phép duyệt dfs(3) kết thúc vậy dfs(1) cũng kết thúc.Và thứ tự duyệt sẽ là: 1, 2, 5, 4, 3.

Ví dụ duyệt theo chiều sâu đồ thị hình I.3 bắt đầu từ đỉnh A: duyệt A, A có các đỉnh kề là B, C, D, theo tứ tự đó thì B đuợc duyệt. B có một đỉnh kề chưa được duyệt là F, nên F được duyệt. F có các đỉnh kề chưa được duyệt là D, G, theo thứ tự đó thì ta duyệt D. D có các đỉnh kề chưa được duyệt là C, E, G, theo thứ tự đó thì C được duyệt. Các đỉnh kề với C đều đã được duyệt nên giải thuật tiếp tục với E. E có một đỉnh kề chưa duyệt là G, vậy ta duyệt G. Lúc này tất cả các đỉnh đều đã được duyệt xong. Vậy thứ tự đỉnh được duyệt là ABFDCEG.



Hình I.3

2. Duyệt theo chiều rộng (breadth-first search)

Giả sử ta có đồ thị G với các đỉnh ban đầu được đánh dấu là chưa duyệt (mảng đánh dấu mang giá trị 0). Từ một đỉnh v nào đó ta bắt đầu duyệt như sau: đánh dấu v đã được duyệt, kế đến là duyệt tất cả các đỉnh kề với v. Khi ta duyệt một đỉnh v rồi đến

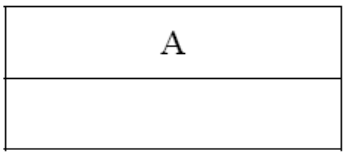
đỉnh w thì các đỉnh kề của v được duyệt trước các đỉnh kề của w, vì vậy ta dùng một hàng đợi để lưu trữ các đỉnh theo thứ tự được duyệt để có thể duyệt các đỉnh kề với chúng. Ta cũng dùng mảng một chiều DX để đánh dấu một đỉnh đã duyệt hay chưa. Giải thuật duyệt theo chiều rộng được viết dạng lặp như sau:

```
//n là số đỉnh của đồ thị
void bfs(vertex v) // v thuoc [1..n], v là đỉnh bắt đầu duyệt
{
    //đánh dấu chưa duyệt tất cả các đỉnh
    for (v = 1; v<=n; v++) DX[v] = 0;
    QUEUE  Q;//sử dụng hàng đơi Q
    vertex x,y;
    DX[v] = 1;
    ENQUEUE(v,Q);
    while !(EMPTY_QUEUE(Q))
    {
        x = DEQUEUE(Q); //lấy x ra khoi Q
        for (mỗi đỉnh y kề với x)
        {
            if (DX[y] ==0)
            {
                DX[y] = 1; //duyệt y
                ENQUEUE(y,Q);
            }
        }
    }
}
```

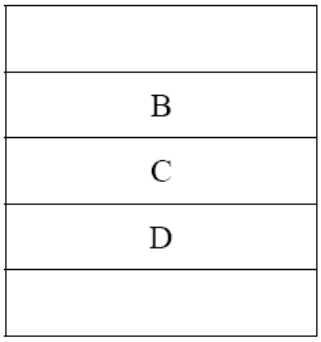
Ví dụ duyệt theo chiều rộng đồ thị hình I.1.a. Giả sử bắt đầu duyệt từ 1. Đỉnh 1 có ba đỉnh kề là 2 và 3. Duyệt 2, 3 và đánh dấu 2, 3 đã duyệt. Tiếp theo là duyệt đến các đỉnh kề với 2 có 5. Duyệt 5 và đánh dấu 5 đã đánh, xét tiếp các đỉnh kề với 3, có đỉnh 4 chưa xét, do đó duyệt 4. Đến đây tất cả các đỉnh đã xét, ta dừng thuật toán. Vậy thứ tự duyệt theo chiều rộng đồ thị hình I.1.a là 1, 2, 3, 5, 4

Ví dụ duyệt theo chiều rộng đồ thị hình I.3. Giả sử bắt đầu duyệt từ A. Duyệt A, kế đến duyệt tất cả các đỉnh kề với A, đó là B, C, D theo thứ tự đó. Kế tiếp duyệt các đỉnh kề của B, C, D theo thứ tự đó. Vậy các đỉnh được duyệt tiếp theo là F, E, G. Có thể minh họa hoạt động của hàng đợi trong phép duyệt trên như sau:

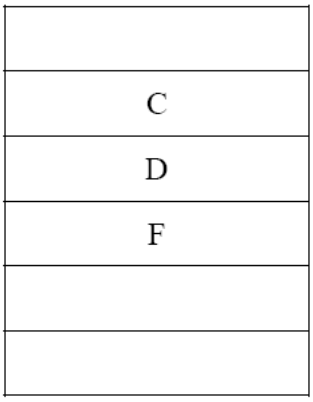
Duyệt A có nghĩa là đánh dấu đã xét và đưa nó vào hàng đợi:



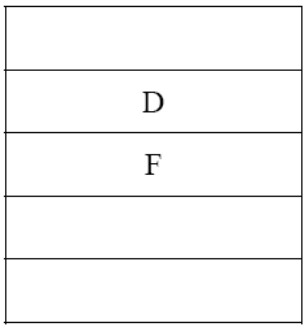
Kế đến duyệt tất cả các đỉnh kề với đỉnh đầu hàng mà chưa được duyệt, tức là ta loại A khỏi hàng đợi, duyệt B, C, D và đưa chúng vào hàng đợi, bây giờ hàng đợi chứa các đỉnh B, C, D.



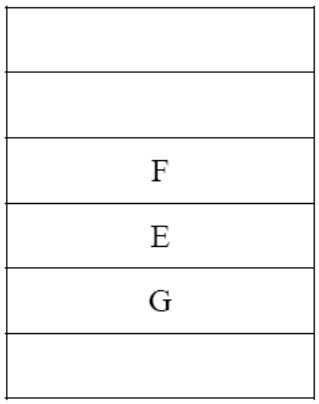
Kế đến lấy B ra khỏi hàng đợi và các đỉnh kề với B mà chưa được duyệt, đó là F, sẽ được duyệt, F được đẩy vào hàng đợi.



Kế đến thì C được lấy ra khỏi hàng đợi và các đỉnh kề với C mà chưa được duyệt sẽ được duyệt. Không có đỉnh nào như vậy, nên bước này không thêm đỉnh nào được duyệt.



Kế đến thì D được lấy ra khỏi hàng đợi và duyệt các đỉnh kề chưa duyệt của D, tức là E, G được duyệt. E, G được đưa vào hàng đợi.



Tiếp tục, F được lấy ra khỏi hàng đợi. Không có đỉnh nào kề với F mà chưa được duyệt. Vậy không duyệt thêm đỉnh nào.

Tương tự như F, E rồi đến G được lấy ra khỏi hàng. Hàng trở thành rỗng và thuật giải kết thúc.

V. Một số bài toán trên đồ thị

Phần này sẽ giới thiệu với một số bài toán quan trọng trên đồ thị, như bài toán tìm đường đi ngắn nhất, bài toán tìm bao đóng chuyển tiếp, cây bao trùm tối thiểu…

1. Bài toán tìm đường đi ngắn nhất từ một đỉnh của đồ thị

Cho đồ thị G với tập các đỉnh V và tập các cạnh E (đồ thị có hướng hoặc vô hướng). Mỗi cạnh của đồ thị có một nhãn, đó là một giá trị không âm, nhãn này còn gọi là giá (cost) của cạnh. Cho trước một đỉnh v xác định, gọi là đỉnh nguồn. Vấn đề là tìm đường đi ngắn nhất từ v đến các đỉnh còn lại của G, tức là các đường đi từ v đến các đỉnh còn lại với tổng các giá (cost) của các cạnh trên đường đi là nhỏ nhất. Chú ý rằng nếu đồ thị có hướng thì đường đi này là có hướng.

Ta có thể giải bài toán này bằng cách xác định một tập hợp S chứa các đỉnh mà khoảng cách ngắn nhất từ nó đến nguồn v đã biết. Khởi đầu S = {v}, sau đó mỗi bước ta sẽ thêm vào S các đỉnh mà khoảng cách từ nó đến v là ngắn nhất. Với giả thiết mỗi cung có một giá trị không âm thì ta luôn luôn tìm được một đường đi ngắn nhất như vậy mà chỉ đi qua các đỉnh đã tồn tại trong S. Để chi tiết hóa thuật giải, giả sử G có n đỉnh và nhãn trên mỗi cung được lưu trong mảng hai chiều C, tức C[i,j] là giá (có thể xem như độ dài) của cung (i,j), nếu i, j không nối nhau thì C[i,j] = ∞ (VC). Ta dùng mảng một chiều L có n phần tử để lưu độ dài của đường đi ngắn nhất từ mỗi đỉnh của

đồ thị đến v. Khởi đầu khoảng cách này chính là độ dài cạnh (v, i), tức là L[i] = C[v,i]. Tại mỗi bước của giải thuật thì L[i] sẽ được cập nhật lại để lưu độ dài đường đi ngắn nhất từ đỉnh v đến đỉnh i, đường đi này chỉ đi qua các đỉnh đã có trong S.

Dưới đây là mô tả giải thuật Dijkstra để giải bài toán trên.

Kí hiệu:

- *L(v): để chỉ nhãn của đỉnh v, tức là cận trên của chiều dài đường đi ngắn nhất từ s_0 đến v.*
- *$d(s_0, v)$: chiều dài đường đi ngắn nhất từ s_0 đến v.*
- *m(s, v): trong số của cạnh (s,v).*

Mô tả

Input: G, s_0

Output: $d(s_0, v)$, với mọi v khác s_0

- *Khởi động:*

  $L(v) = \infty, \forall v \neq s_0$; //nhãn tạm thời

  S = Rỗng;

- *Bước 0*

  $d(s_0, s_0) = L(s_0) = 0$;

  $S = \{s_0\}$; //s_0 có nhãn chính thức

- *Bước 1*

  - *Tính lại nhãn tạm thời L(v), với $v \notin S$*

    Nếu v kề với s_0 thì

    $L(v) = Min\{L(v), L(s_0) + m(s_0, v)\}$;

  - *Tìm $s_1 \notin S$ và kề với s_0 sao cho:*

    $L(s_1) = Min\{L(v): \forall v \notin S\}$; // khi đó $d(s_0, s_1) = L(s_1)$

  - *$S = S \cup \{s_1\}$; //$S = \{s_0, s_1\}$, s_1 có nhãn chính thức*

- *Bước 2*

  - *Tính lại nhãn tạm thời L(v), với $v \notin S$*

    Nếu v kề với s_1 thì

    $L(v) = Min\{L(v), L(s_1) + m(s_1, v)\}$;

  - *Tìm $s_2 \notin S$ và kề với s_1 sao cho:*

    $L(s_2) = Min\{L(v): \forall v \notin S\}$; // khi đó $d(s_0, s_2) = L(s_2)$

  Nếu $L(s_2) = Min\{L(s_j), L(s_j)+m(s_j, s_2)\}$ thì đường đi từ s_0 đến s_2 qua s_j là bé nhất, và s_j là đỉnh kề trước s_2

- $S = S \cup \{s_2\}$; //$S = \{s_0, s_1, s_2\}$, s_2 có nhãn chính thức

- ...
- Bước i
  - Tính lại nhãn tạm thời L(v), với $v \notin S$

    Nếu v kề với s_{i-1} thì

    $L(v) = Min\{L(v), L(s_{i-1}) + m(s_{i-1}, v)\}$;
  - Tìm $s_i \notin S$ và kề với $s_j, j \in [0,i-1]$ sao cho:

    $L(s_i) = Min\{L(v): \forall v \notin S\}$; // khi đó $d(s_0, s_i) = L(s_i)$

    Nếu $L(s_i) = Min\{L(s_j), L(s_j)+m(s_j, s_i)\}$ thì đường đi từ s_0 đến s_i qua s_j là bé nhất, và s_j là đỉnh kề trước s_i
  - $S = S \cup \{s_2\}$; //$S = \{s_0, s_1, s_2...s_i\}$, s_i có nhãn chính thức

Cài đặt thuật toán Dijkstra

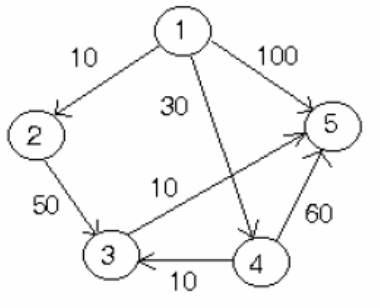
Để cài đặt thuật giải dễ dàng, ta giả sử các đỉnh của đồ thị được đánh số từ 1 đến n, tức là V = {1,..,n} và đỉnh nguồn là 1. Nếu muốn lưu trữ lại các đỉnh trên đường đi ngắn nhất để có thể xây dựng lại đường đi này từ đỉnh nguồn đến các đỉnh khác, ta dùng một mảng ddnn. Mảng này sẽ lưu ddnn[u] = w với u là đỉnh "trước" đỉnh w trong đường đi.

```
void Dijkstra(int v, int  C[max][max])
{
   int dnnn[Max];//mảng chứa đường đi ngắn nhất
   int i, k, min, dht; //dht: đỉnh hiện tại
   int DX[Max]; //đánh dấu các đỉnh đã đưa vào S
   int L[Max]; //L[i] chứa chi phí tới đỉnh i

   for (i =1; i<=SoDinh; i++)
   {
      DX[i] = 0;
      L[i] = VC; //VC: vô cùng
   }
   DX[v] = 1;
   L[v] = 0;
   dht = v;
   int h = 1;
```

```
while(h<SoDinh-1)
{
    min = CV;
    for(int i=1; i<=SoDinh; i++)
    {
        if(DX[i] == 0)
        {
            if(L[dht] + C[dht][i] < L[i]) //tính lại nhãn
            {
                L[i] = L[dht] + C[dht][i];
                dnnn[i] = dht; // gán đỉnh hiện tại bằng đỉnh
                                trước i trên lộ trình
            }
            if(L[i] < min) // chọn đỉnh k
            {
                min = L[i];
                k = i;
            }
        }
        //Tại mỗi bước lặp h, tìm được đường đi ngắn nhất từ s
        //đến k
        Xuatddnn(v,k, ddnn);
        cout<<"\nTrong so: "<< L[k];
        dht = k; // khởi động lại dht
        DX[dht] = 1; //Đưa nút k vào tập nút đã xét
        h++;
    }
  }
}
```

Ví dụ: áp dụng thuật giải Dijkstra cho đồ thị hình I.5



Hình I.5

Kết quả khi áp dụng giải thuật

Lần lặp	S	W	L[2]	L[3]	L[4]	L[5]
Khởi đầu	{1}	-	**10** **(1)**	∞	30 (1)	100 (1)
1	{1,2}	2	10 (1)	60 (2)	**30** **(1)**	100 (1)
2	{1,2,4}	4	10 (1)	**40** **(4)**	30 (1)	90 (4)
3	{1,2,3,4}	3	10 (1)	40 (4)	30 (1)	**50** **(3)**
4	{1,2,3,4,5}	5	10 (1)	40 (4)	30 (1)	50 (3)

Mảng ddnn có giá trị như sau:

1	2	3	4	5
	1	4	1	3

Từ kết quả trên ta có thể suy ra rằng đường đi ngắn nhất từ đỉnh 1 đến đỉnh 3 là 1→4→3 có độ dài là 40. Đường đi ngắn nhất từ 1 đến 5 là 1→4→3→5 có độ dài là 50.

Bài tập:

1. Viết thủ tục xuất đường đi Xuatdnnn(int v, int k, int ddnn[max]).
2. Cài đặt thuật giải Dijkstra.

2. Bài toán tìm bao đóng chuyển tiếp.

Trong một số trường hợp ta chỉ cần xác định có hay không một đường đi nối giữa hai đỉnh i,j bất kì. Bây giờ khoảng cách giữa i, j là không quan trọng mà ta chỉ cần biết i,j được nối với nhau bởi một cạnh, ngược lại C[i,j]=0 (có nghĩa là false). Lúc này mảng

A[i,j] không cho khoảng cách ngắn nhất giữa i, j mà nó cho biết là có đường đi từ i đến j hay không. A gọi là bao đóng chuyển tiếp trong đồ thị G có biểu diễn ma trận kề là C. Giải thuật tìm bao đóng chuyển tiếp hay còn gọi giải thuật Warshall.

```
int A[n,n], C[n,n]; //A là bao đóng chuyển tiếp, C là ma trận kề
void Warshall()
{
    int i,j,k;
    for (i=1; i<=n; i++)
        for (j=1; j<=n; j++)
            A[i-1,j-1] = C[i-1,j-1];
    for (k=1; k<=n; k++)
        for (i=1; i<=n; i++)
            if(A[i,k] != 0)
                for (j=1; j<=n; j++)
                    if(A[k][j])
                        A[i,j] = 1;
}
```

3. Bài toán tìm cây bao trùm tối thiểu (minimum-cost spanning tree)

Giả sử ta có một đồ thị vô hướng G = (V, E). Đồ thị G gọi là liên thông nếu tồn tại đường đi giữa hai đỉnh bất kì. Bài toán tìm cây bao trùm tối thiểu (hoặc cây phủ tối thiểu) là tìm một tập hợp T chứa các cạnh của một đồ thị liên thông C sao cho V cùng với tập các cạnh này cũng là một đồ thị liên thông, tức là (V, T) là một đồ thị liên thông. Hơn nữa tổng độ dài của các cạnh trong T là nhỏ nhất. Một thể hiện của bài toán này trong thực tế là bài toán thiết lập mạng truyền thông, ở đó các đỉnh là các thành phố còn các cạnh của cây bao trùm là đường nối mạng giữa các thành phố.

Giả sử G có n đỉnh được đánh số từ 1..n. Giải thuật Prim để giải bài toán này như sau:

Ý tưởng

- Bắt đầu, tập khởi tạo là U bằng 1 đỉnh nào đó, đỉnh 1 chẳng hạn, U = {1}, T = U.
- Sau đó ta lặp lại cho đến khi U = V, tại mỗi bước lặp ta chọn cạnh nhỏ nhất (u,v) sao cho u $\in$ U, v $\in$ V-U. Thêm v vào U và (u, v) vào T. Khi thuật giải kết thúc thì (U,T) là một cây phủ tối tiểu.

Mô tả thuật toán

- *Input: G=(V,E)*
- *Output: T = (V, ?) là nhỏ nhất.*

- *Khởi động:*
  - *$U \subset V$*
  - *T = (U,.) = Rỗng; //đồ thị rỗng*
  - *U = {1};*
- *Trong khi ($U \neq V$)*

  Tìm cạnh (u,v) có trọng số nhỏ nhất với $u \in U$, $v \in V$. Thêm đỉnh v này vào U, thêm (u,v) vào T

Cài đặt

Để tiến hành cài đặt thuật toán, ta cần mô tả dữ liệu. Đồ thị có trọng số được biểu diễn thành một ma trận kề C[n,n].

Khi tìm cạnh có trọng số nhỏ nhất nối một đỉnh trong U và một đỉnh ngoài U tại mỗi bước, ta dùng hai mảng để lưu trữ:

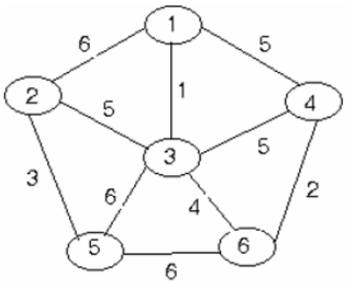
- Mảng closest[], với i $\in$ V\U thì closest[i] $\in$ U là đỉnh kề gần i nhất.
- Mảng lowcost[i] lưu trọng số của cạnh (i, closest[i])
- Mảng daxet đánh đấu đỉnh đã được xét chưa

Tại mỗi bước ta duyệt mảng lowcost để tìm đỉnh closest[k] $\in$ U sao cho trọng số (k, closest[k]) = lowcost[k] là nhỏ nhất. Khi tìm được, ta in cạnh (closest[k],k), cập nhật vào các mảng closest và lowcost, và có k thêm vào U. Khi ta tìm được một đỉnh k cho cây bao trùm, ta cho daxet[k] = DX là đánh dấu đã xét.

```
#define VC 10000 //định nghĩa giá trị vô cùng
#define DX 1 //định nghĩa giá trị khi đỉnh đã được xét
...
void Prim(int C[max][max])
{
    double lowcost[Max];
    int closest[Max];
    int daxet[Max];
    int i,j,k,Min;
    //bắt đầu từ đỉnh số 1
    for(i=2; i<=n; i++)
    {
        lowcost[i] = C[1][i];
        closest[i] = 1;
        daxet[i] = 0;
```

```
        }
        for(i=2; i<=n; i++)
        {
            Min = lowcost[2];
            k = 2;
            for(j=3; j<=n; j++)
            {
                if(!daxet[j] && lowcost[j] < Min)
                {
                    Min = lowcost[j];
                    k = j;
                }
            }
            daxet[k] = DX;
            //Khởi động lại chosest[], lowcost[]
            for(j=2; j<=n; j++)
                if(c[k][j]<lowcost[j] && !daxet[j])
                {
                    lowcost[j] = c[k][j];
                    closest[j] = k
                }
        }
    }
```

Ví dụ: áp dụng giải thuật Prim để tìm cây bao trùm tối thiểu của đồ thị liên thông hình I.6



Hình I.6

Ma trận kề:

	1	2	3	4	5	6
1	0	6	1	5	VC	VC
2	6	0	5	VC	3	VC
3	1	5	0	5	6	4
4	5	VC	5	0	VC	2
5	VC	3	6	VC	0	6
6	VC	VC	4	2	6	0

Khởi tạo

Mảng lowcost

2	3	4	5	6
6	1	5	VC	VC

Mảng closest

2	3	4	5	6
1	1	1	1	1

Mảng daxet

2	3	4	5	6
0	0	0	0	0

Bước 1: tìm được Min = 1, k = 3, mảng lowcost và closest cập nhật như sau:

Mảng lowcost

2	3	4	5	6
5	1	5	6	4

Mảng closest

2	3	4	5	6
3	1	1	3	3

Mảng daxet

2	3	4	5	6
0	1	0	0	0

Bước 2: tìm được Min = 4, k = 6

Mảng lowcost

2	3	4	5	6
5	1	2	6	4

Mảng closest

2	3	4	5	6
3	1	6	3	3

Mảng daxet

2	3	4	5	6
0	1	0	0	1

Bước 3: tìm được Min = 2, k = 4

Mảng lowcost

2	3	4	5	6
5	1	2	6	4

Mảng closest

2	3	4	5	6
3	1	6	3	3

Mảng daxet

2	3	4	5	6
0	1	1	0	1

Bước 4: tìm được Min = 5, k = 2

Mảng lowcost

2	3	4	5	6
5	1	2	3	4

Mảng closest

2	3	4	5	6
3	1	6	2	3

Mảng daxet

2	3	4	5	6
1	1	1	0	1

Bước 5: tìm Min = 3, k = 5

Mảng lowcost

2	3	4	5	6
5	1	2	3	4

Mảng closest

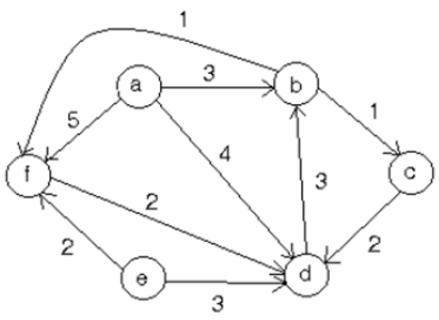
2	3	4	5	6
3	1	6	2	3

Mảng daxet

2	3	4	5	6
1	1	1	1	1

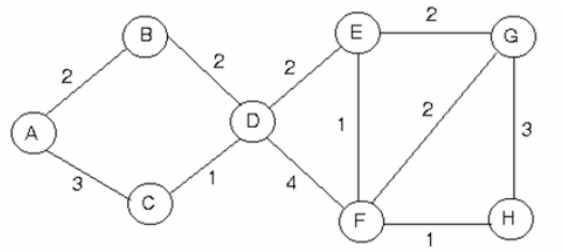
Bài tập

1. Viết biểu diễn đồ thị I.7 bằng:
   - Ma trận kề.
   - Danh sách các đỉnh kề.



Hình I.7

2. Duyệt đồ thị hình I.7 (xét các đỉnh theo thứ tự a,b,c...)
   - Theo chiều rộng bắt đầu từ a.
   - Theo chiều sâu bắt đầu từ f
3. Áp dụng giải thuật Dijkstra cho đồ thị hình I.7, với đỉnh nguồn là a
4. Viết biểu diễn đồ thị I.8 bằng:



Hình I.8

- Ma trận kề.
- Danh sách các đỉnh kề.

5. Duyệt đồ thị hình I.8 (xét các đỉnh theo thứ tự A,B,C...)
   - Theo chiều rộng bắt đầu từ A.
   - Theo chiều sâu bắt đầu từ B.
6. Áp dụng giải thuật Dijkstra cho đồ thị hình I.8, với đỉnh nguồn là A.
7. Tìm cây bao trùm tối thiểu của đồ thị hình I.8 bằng giải thuật Prim.
8. Cài đặt đồ thị có hướng bằng ma trận kề rồi viết các giải thuật:
   - Duyệt theo chiều rộng.
   - Duyệt theo chiều sâu.
   - Tìm đường đi ngắn nhất từ một đỉnh cho trước (Dijkstra).
9. Cài đặt đồ thị có hướng bằng danh sách các đỉnh kề rồi viết các giải thuật duyệt theo chiều rộng.