TRƯỜNG ĐẠI HỌC ĐÀ LẠT
KHOA CÔNG NGHỆ THÔNG TIN

NGUYỄN THỊ THANH BÌNH

BÀI GIẢNG TÓM TẮT

CẤU TRÚC DỮ LIỆU VÀ THUẬT GIẢI 2

Dành cho sinh viên ngành công nghệ thông tin

(Lưu hành nội bộ)

Đà Lạt 2008

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

Tài liệu 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 nó nhằm giúp các bạn sinh viên chuyên ngành có một tài liệu cô đọng dùng làm tài liệu học tập.

Mặc dù đã rất cố gắng nhiều trong quá trình biên soạn giáo trình, song 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 06 năm 2008

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

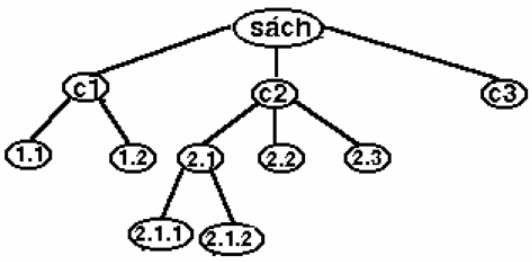
Cây là một tập hợp các phần tử gọi là nút (nodes) trong đó có một nút được phân biệt gọi là nút gốc (root). Trên tập hợp các nút này có một quan hệ, gọi là mối quan hệ cha - con (parenthood), để xác định hệ thống cấu trúc trên các nút. Mỗi nút, trừ nút gốc, có duy nhất một nút cha. Một nút có thể có 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.

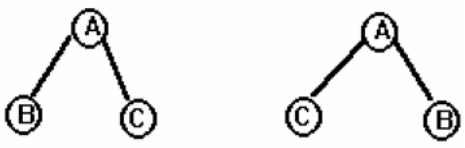
Nếu có đường đi từ nút a đến nút b thì ta nói a là tiền bối (ancestor) của b, còn b gọi là hậu duệ (descendant) của nút a. Rõ ràng một nút vừa là tiền bối vừa là hậu duệ của chính nó. Tiền bối hoặc hậu duệ của một nút khác với chính nó gọi là tiền bối hoặc hậu duệ thực sự. Trên cây nút gốc không có tiền 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

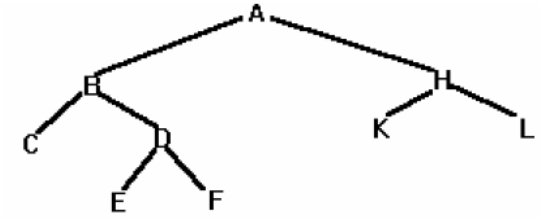
Trong trường hợp ta không phân biệt rõ ràng thứ tự các nút thì ta gọi là cây không có thứ tự. Các nút con cùng một nút cha gọi là các nút anh em ruột (siblings). Quan hệ "trái sang phải" của các anh em ruột có thể mở rộng cho hai nút bất kỳ theo qui tắc: nếu a, b là hai anh em ruột và a bên trái b thì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

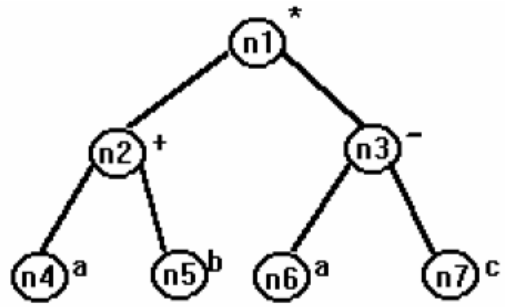
hậu tự: C E F D B K L H A

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

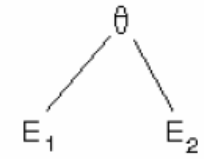
chứa một record về thông tin của sinh viên (mã SV, họ tên, ngày sinh, địa chỉ,...) thì khoá có thể là mã SV hoặc họ tên hoặc ngày sinh tuỳ theo giá trị nào ta đang quan tâm đến trong giải thuật.

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).

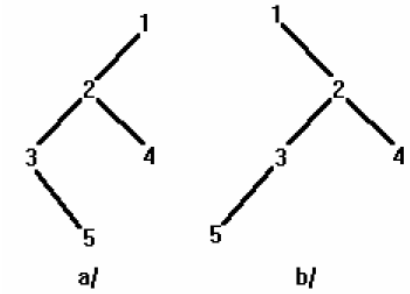
Hình I.6: Cây biểu diễn biểu thức a+b và b+a

- 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/.

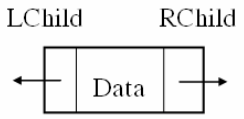
2. Vài tính chất của cây nhị phân

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>=$\log_2(n+1)$

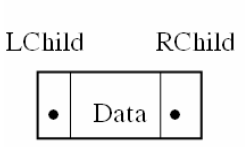
3. Biểu diễn cây nhị phân

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.

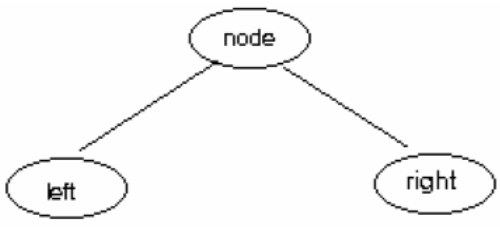
Nút lá có dạng



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):

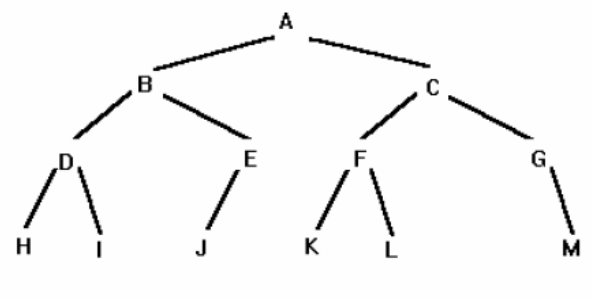
- Duyệt tiền tự (Node-Left-Right): duyệt nút gốc, duyệt tiền tự con trái rồi duyệt tiền tự con phải.
- Duyệt trung tự (Left-Node-Right): duyệt trung tự con trái rồi đến nút gốc sau đó là duyệt trung tự con phải.
- Duyệt hậu tự (Left-Right-Node): duyệt hậu tự con trái rồi duyệt hậu tự con phải sau đó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

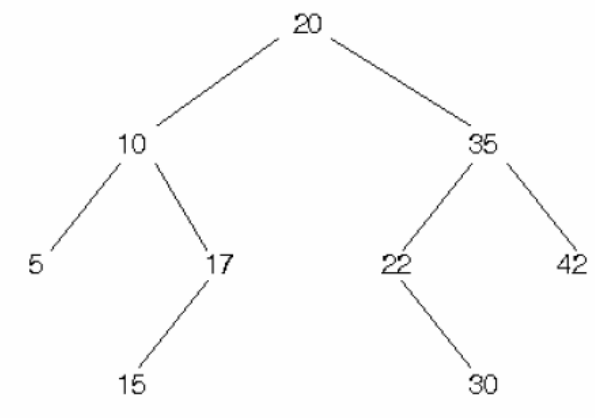
IV. Cây tìm kiếm nhị phân (Binary Search Trees)

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.

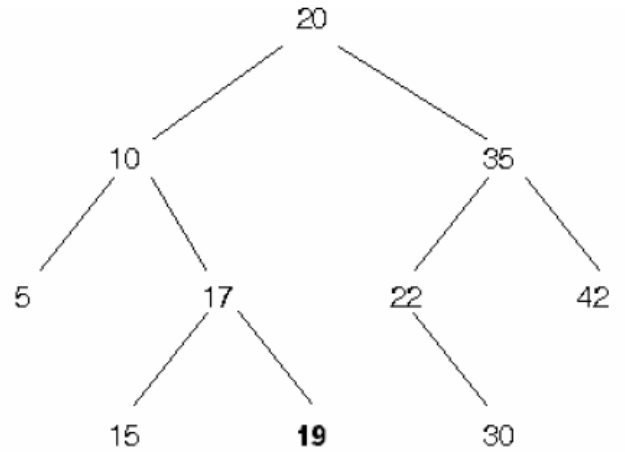
Việc thêm một khoá vào cây TKNP là việc tìm kiếm và thêm một nút, tất nhiên, phải đảm bảo cấu trúc cây TKNP không bị phá vỡ. Giải thuật cụ thể như sau:

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;
```

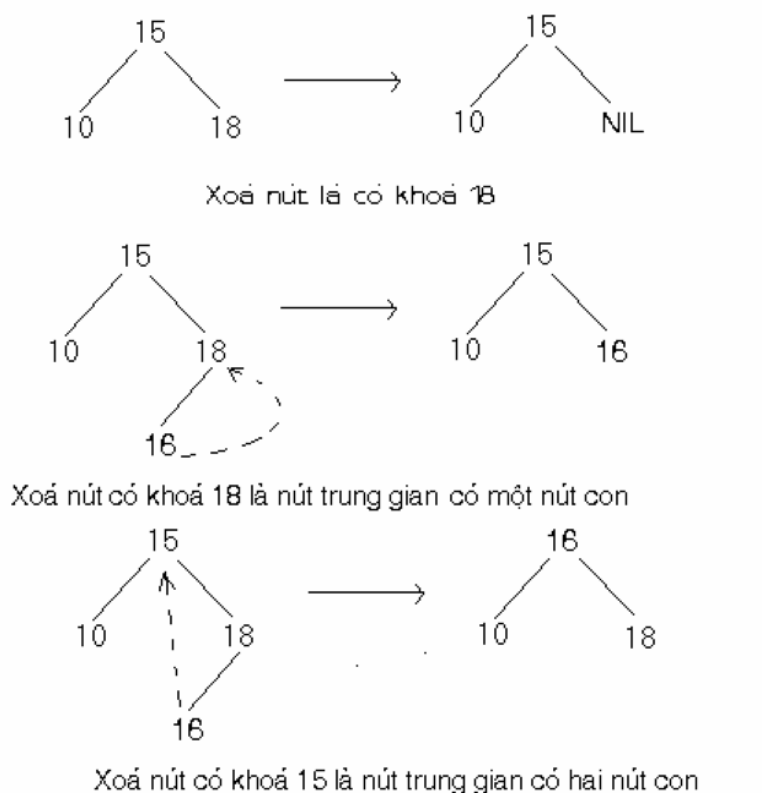
```
            Root->Key = x;
            Root->left = NULL;
            Root->right = NULL;
    }
    else if (x < Root->Key) InsertNode(x,Root->left);
    else if (x>Root->Key)InsertNode(x,Root->right);
}
```

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

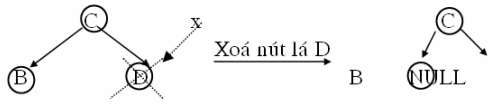
Giả sử ta muốn xoá một nút có khoá x, trước hết ta phải tìm kiếm nút chứa khoá x trên cây.



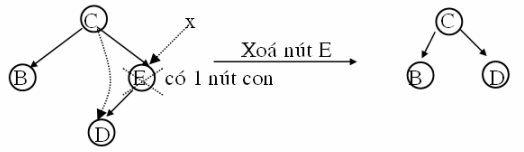
Hình I.12

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.

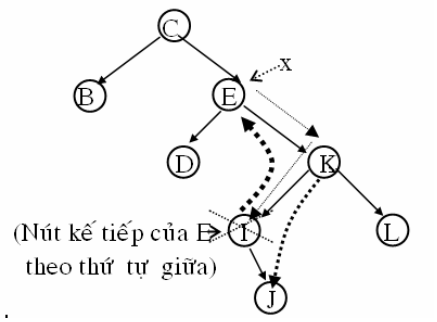


  - N chỉ có một nút con ta thay nó bởi nút con của nó.



  - 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;
    }
}
```

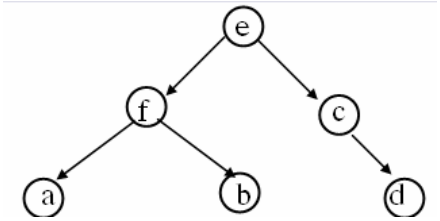
V. Cây nhị phân tìm kiếm cân bằng (Cây AVL)

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)

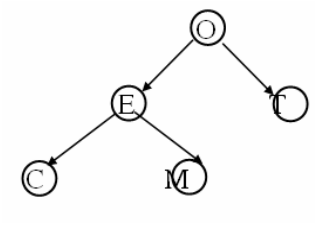
Trên cây nhị phân tìm kiếm BST có n phần tử mà là cây CBHT, phép tìm kiếm một phần tử trên nó sẽ thực hiện rất nhanh: trong trường hợp xấu nhất, ta chỉ cần thực hiên $\log_2 n$ phép so sánh. Nhưng cây CBHT có cấu trúc kém ổn định trong các thao tác cập nhật cây, nên nó ít được sử dụng trong thực tế. 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	$\Leftrightarrow$	$h_R(p) = h_L(p)$:2 cây con cao bằng nhau
CSCB(p) = RH	$\Leftrightarrow$	$h_R(p) > h_L(p)$: cây lệch phải
CSCB(p) = LH	$\Leftrightarrow$	$h_R(p) < h_L(p)$: 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:

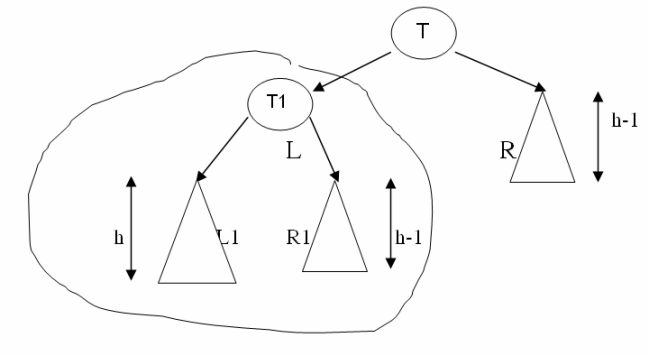
```
typedef    .....   ElementType;  /* Kiểu dữ liệu của nút */
typedef  struct  AVLTN
{
    ElementType Data;
    int  Balfactor; //Chỉ số cân bằng
    struct AVLTN * Lchild, *Rchild;
} AVLTreeNode;
typedef   AVLTreeNode  *AVLTree;
```

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

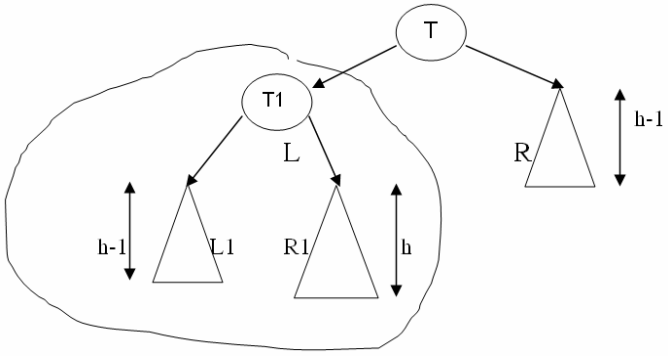
Ngoài các thao tác thêm và hủy đối với cây cân bằng, ta còn có thêm thao tác cơ bản là cân bằng lại cây AVL trong trường hợp thêm hoặc hủy một nút của nó. Khi đó độ lệch giữa chiều cao cây con phải và trái sẽ là 2. Do đó trường hợp cây lệch trái và phải tương ứng là đối xứng nhau, nên ta chỉ xét trường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

Trường hợp a: cây con T1 lệch trái



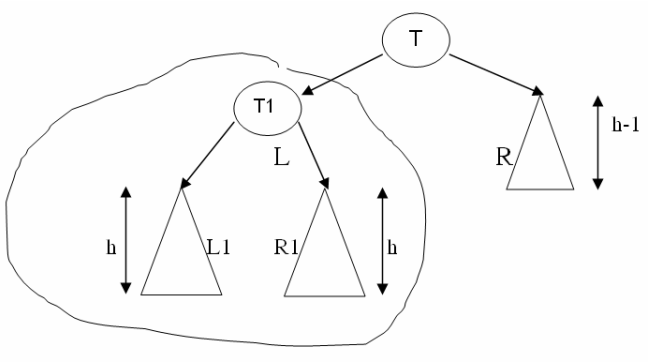
Hình I.15

Trường hợp b: cây con T1 lệch phải



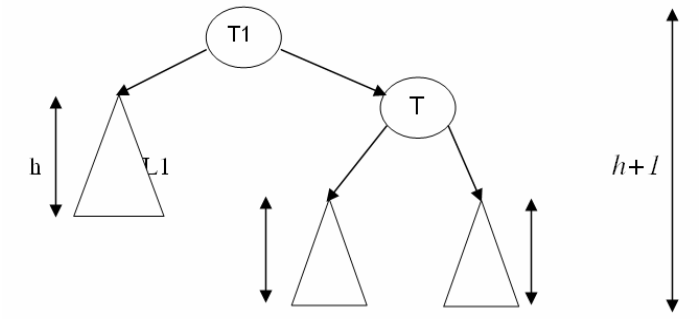
Hình I.16

Trường hợp c: cây con T1 không lệch



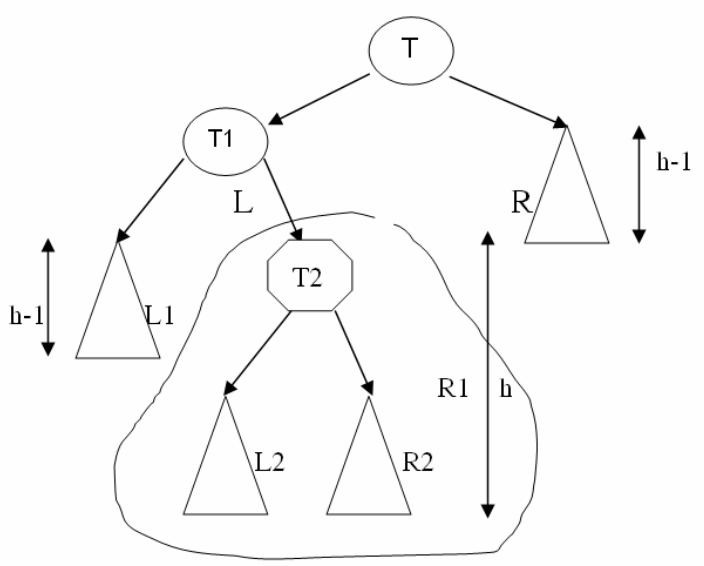
Hình I.17

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



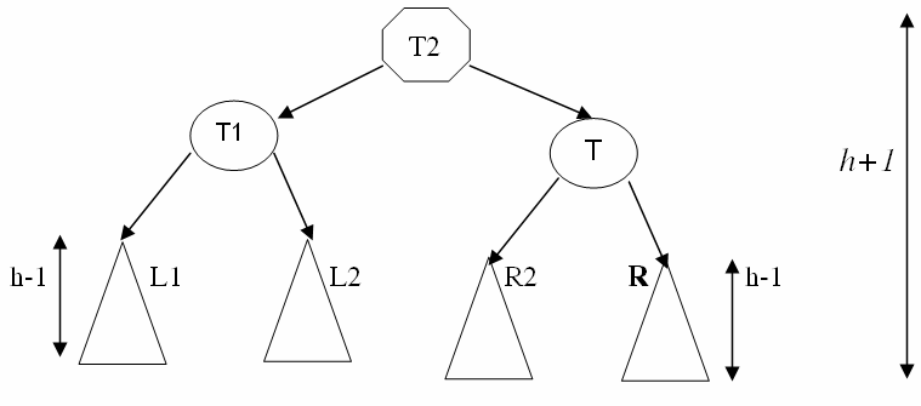
Hình I.18

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.

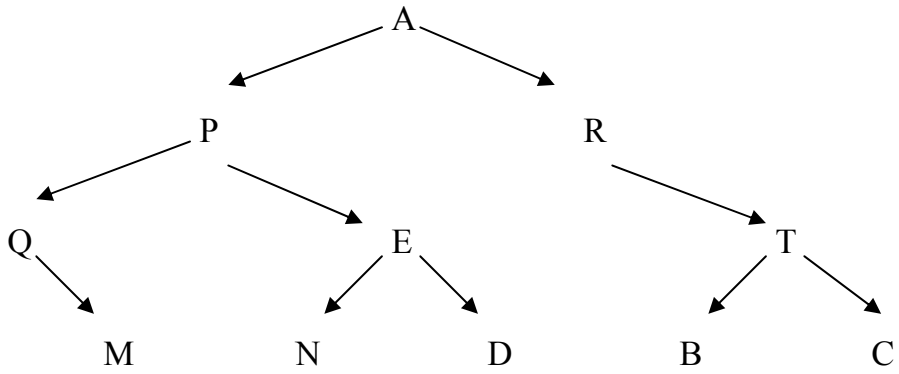
```
int DeleteAVL(AVLTree &T, ElementType x)
{
    int Kqua;
    if (T== NULL) return 0; // không có x trên cây
    if  (T-> Data >  x)
    {
        Kqua = DeleteAVL(T->Lchild,x);// tìm và xóa x trên cây con trái của T
        if (Kqua < 2) return  Kqua;
        switch  (T->Balfactor)
        {
            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 bắng 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.

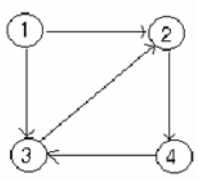
Nội dung

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

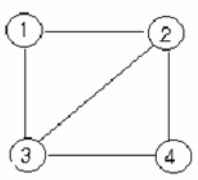
- Các khái niệm cơ bản
- Kiểu dữ liệu trừu tượng đồ thị
- Biểu diễn đồ thị
- Các phép duyệt đồ thị
- Một số bài toán trên đồ thị

I. Các định nghĩa

Một đồ thị G bao gồm một tập hợp V các đỉnh và một tập hợp E các cung tương ứng, kí hiệu G = (V, E). Các đỉnh còn được gọi là nút (node) hay điểm (point). Các cung nối giữa hai đỉnh, hai đỉnh này có thể trùng nhau. Hai đỉnh có cung nối nhau gọi là hai đỉnh kề (adjacency). 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 thì G gọi là đồ thị có hướng (directed graph). Nếu các cung trong đồ thị G không có thứ tự thì đồ thị G gọi là đồ thị vô hướng (undirected graph). Trong các phần sau này ta dùng từ đồ thị để nói đến đồ thị nói chung, khi nào cần phân biệt rõ ta sẽ dùng đồ thị có hướng hay đồ thị vô hướng. Hình I.1a cho ta một ví dụ về đồ thị có hướng, hình I.1.b cho ta ví dụ về đồ thị vô hướng. Trong các đồ thị này thì các vòng tròn được đánh số biểu diễn các đỉnh, còn các cung được biểu diễn bằng đoạn thẳng có hướng (trong I.1a) hoặc không có hướng (trong I.1b).



Hình I.1.a



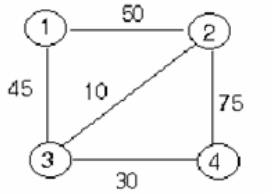
Hình I.1.b

Thông thường trong một đồ thị, các đỉnh biểu diễn cho các đối tượng còn các cung biểu diễn cho mối quan hệ giữa các đối tượng đó. Chẳng hạn các đỉnh có thể biểu diễn cho các thành phố còn các cung biểu diễn cho đường giao thông nối giữa các thành phố.

Một đường đi (path) trên đồ thị là một dãy tuần tự các đỉnh v_1, v_2, ..v_n sao cho (v_i, v_{i+1}) là một 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, 4 trong đồ thị I.1.a là một đường đi từ đỉnh 1 đến đỉnh 4, 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 1,3,4,2,1 là một chu trình có độ dài bằng 4.

Trong nhiều ứng dụng ta thường kết hợp các giá trị (value) hay nhãn (label) 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 cho ta một ví dụ về một đồ thị có nhãn. Ở đây, nhãn là các giá trị số nguyên biểu diễn cho giá cước vận chuyển một tấn hàng giữa các thành phố 1, 2, 3, 4 chẳng hạn.



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 tất cả các cạnh (v, w) E sao cho v, w $\in$ V’

III. Biểu diễn đồ thị

Một số cấu trúc dữ liệu có thể dùng để biểu diễn đồ thị. Việc chọn cấu trúc dữ liệu nào là tuỳ thuộc vào các phép toán trên các cung và đỉnh của đồ thị. Hai cấu trúc thường

gặp là biểu diễn đồ thị bằng ma trận kề (adjacency matrix) và biểu diễn đồ thị bằng danh sách các đỉnh kề (adjacency list).

1. Biểu diễn đồ thị bằng ma trận kề

Ta dùng một mảng hai chiều, chẳng hạn mảng A, kiểu boolean để biểu diễn các đỉnh kề. Nếu đồ thị có n đỉnh thì ta dùng mảng A kích thước n x n. Giả sử các đỉnh được đánh số 1..n thì A[i,j] = true, nếu có cạnh nối giữa hai đỉnh i và j, ngược lại A[i,j] = false. Rõ ràng nếu đồ thị G là đồ thị có 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	0	1	2	3
0	true	true	true	**false**
1	true	true	true	true
2	true	true	true	true
3	**false**	true	true	true

Ta cũng có thể biểu diễn true là 1 còn false là 0. Với cách biểu diễn này thì đồ thị hình I.1a có biểu diễn ma trận kề như sau:

j / i	0	1	2	3
0	1	1	1	0
1	0	1	0	1
2	0	1	1	0
3	0	0	0	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 A[i,j] = a. Ví dụ ma trận kề của đồ thị hình I.2 là:

j / i	1	2	3	4
1		50	45	
2	50		10	75
3	45	10		30
4		75	30	

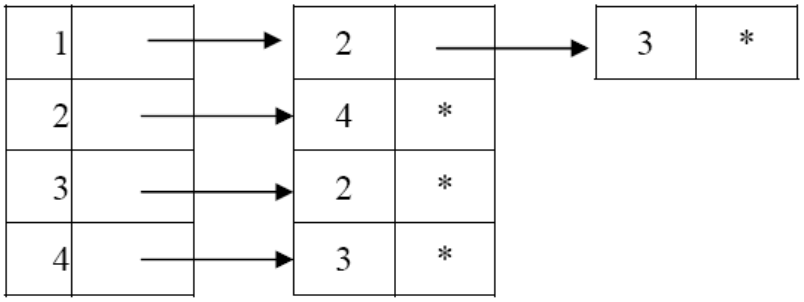
Ở đây các cặp đỉnh không có cạnh nối thì ta để trống, nhưng trong các ứng dụng ta có thể phải gán cho nó một giá trị đặc biệt nào đó để phân biệt với các giá trị có nghĩa khác. Chẳng hạn như trong bài toán tìm đường đi ngắn nhất, các giá trị số nguyên biểu diễn cho khoảng cách giữa hai thành phố thì cặp thành phố không có cạnh nối ta gán cho nó khoảng cách bằng μ, 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 HEAD một chiều có n phần tử để biểu diễn cho đồ thị có n đỉnh. HEAD[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 HEAD

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.

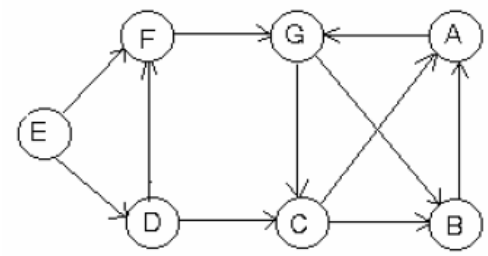
1. Duyệt theo chiều sâu (Depth-first search)

Giả sử ta có đồ thị G = (V, E) với các đỉnh ban đầu được đánh dấu là chưa duyệt (unvisited). 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 mark 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 =0; v <n; v++) mark[v]=unvisited;
//duyệt theo chiều sâu từ đỉnh đánh số 0
for (v = 0; v<n; v++)
    if (mark[v] == unvisited)
            dfs(v); //duyệt theo chiều sâu đỉnh v
```

Thủ tục dfs ở trong giải thuật trên có thể được viết như sau:

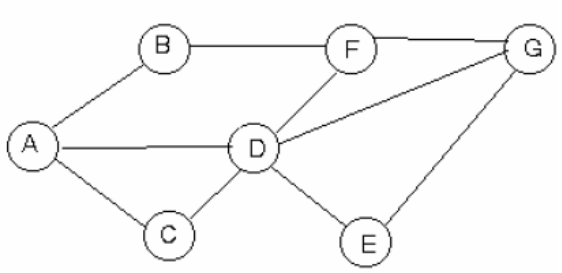
```
void dfs(vertex v) // v thuộc [0..n]
{
    vertex w;
    mark[v]=visited;
    for (mỗi đỉnh w là đỉnh kề với v)
    if (mark[w] == unvisited)
    dfs(w);
}
```



Hình I.3

Ví dụ: duyệt theo chiều sâu đồ thị trong hình I.3. Giả sử ta bắt đầu duyệt từ đỉnh A, tức là dfs(A). Giải thuật sẽ đánh dấu A đã được duyệt, rồi chọn đỉnh đầu tiên trong danh sách các đỉnh kề với A, đó là G. Tiếp tục duyệt G, G có hai đỉnh kề là B và C, theo thứ tự đó thì đỉnh kế tiếp được duyệt là đỉnh B. B có một đỉnh kề là A, nhưng A đã được đánh dấu đã duyệt nên phép duyệt dfs(B) đã hoàn tất. Bây giờ giải thuật sẽ tiếp tục với đỉnh kề với G mà còn chưa duyệt là C. C không có đỉnh kề nên phép duyệt dfs(C) kết thúc vậy dfs(A) cũng kết thúc. Còn lại 3 đỉnh chưa được duyệt là D, E, F.

Ví dụ duyệt theo chiều sâu đồ thị hình I.4 bắt đầu từ đỉnh A: duyệt A, A có các đỉnh kề là B, C, D, theo tứ tự đó thì B được 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 node đều đã được duyệt xong. Vậy thứ tự đỉnh được duyệt là ABFDCEG.



Hình I.4

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 (unvisited). 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 node theo thứ tự được duyệt để có thể duyệt các đỉnh kề với chúng. Ta cũng dùng mảng một chiều mark để đánh dấu một node đã duyệt hay chưa, tương tự như duyệt theo chiều sâu. Giải thuật duyệt theo chiều rộng được viết như sau:

```
//đánh dấu chưa duyệt tất cả các đỉnh
for (v = 0; v<n; v++) mark[v] = unvisited;
//n là số đỉnh của đồ thị
//duyệt theo chiều rộng từ đỉnh đánh số 0
for (v = 0; v<n; v++)
    if (mark[v] == unvisited)
            bfs(v);
```

Thủ tục bfs viết như sau:

```
void bfs(vertex v) // v thuoc [1..n]
{
    QUEUE of vertex Q;
    vertex x,y;
    mark[v] = visited;
    ENQUEUE(v,Q); //push cac dinh ke voi v vao Q
    while !(EMPTY_QUEUE(Q))
    {
            x = Pop(Q); //lấy x ra khoi Q
            for (mỗi đỉnh y kề với x)
            {
                    if (mark[y-1] == unvisited)
```

```
                {
                    mark[y-1] = visited; {duyệt y}
                    ENQUEUE(y,Q);
                }
            }
        }
}
```

Ví dụ duyệt theo chiều rộng đồ thị hình I.3. Giả sử bắt đầu duyệt từ A. A chỉ có một đỉnh kề G, nên ta duyệt G. Kế đến duyệt tất cả các đỉnh kề với G, đó là B, C. Sau đó duyệt tất cả các đỉnh kề với B, C theo thứ tự đó. Các đỉnh kề với B, C đã được duyệt, nên ta tiếp tục duyệt các đỉnh chưa được duyệt. Các đỉnh chưa được duyệt là D, E, F. Duyệt D, kế đến duyệt E, cuối cùng duyệt F. Vậy thứ tự các đỉnh được duyệt là: AGBCDFE.

Ví dụ duyệt theo chiều rộng đồ thị hình I.4. 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 node 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 visited và đưa nó vào hàng đợi:

A

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.

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.

C
D
F

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.

D
F

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.

F
E
G

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

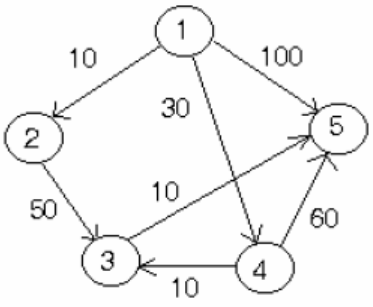
Để cài đặt thuật giải dễ dàng, ta giả sử các đỉnh của đồ thị được đánh số từ 1 đến n, tức là V = {1,..,n} và đỉnh nguồn là 1. Nếu muốn lưu trữ lại các đỉnh trên đường đi ngắn nhất để có thể xây dựng lại đường đi này từ đỉnh nguồn đến các đỉnh khác, ta dùng một mảng ddnn. Mảng này sẽ lưu ddnn[u] = w với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

```
void Dijkstra(int v)
{
   int 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<=n; i++)
   {
      DX[i] = 0;
      L[i] = VC; //VC: vô cùng
   }
   DX[v] = 1;
   L[v] = 0;
   dht = v;
   int h = 1;
   while(h<n-1)
   {
      min = CV;
      for(int i=1; i<=n; 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])

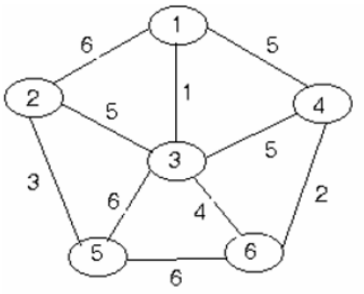
- Mảng daxet đánh đấu đỉnh đã được xét chưa

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(MaTranKe C)
{
    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

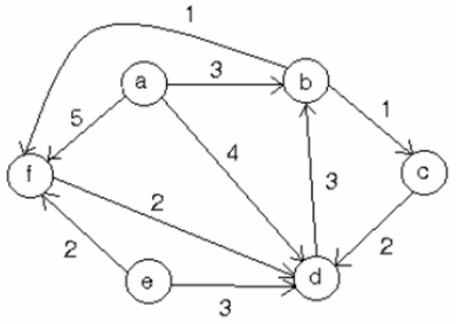
Mảng daxet

2	3	4	5	6

1	1	1	1	1

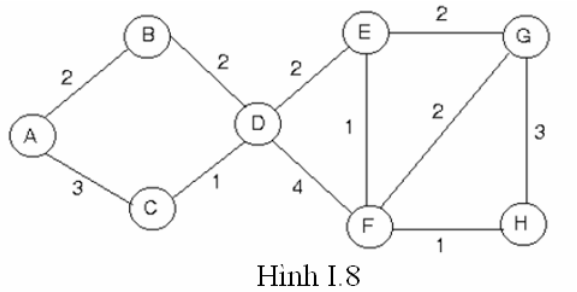
Bài tập

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.

Chương III

Bảng Băm

Mục tiêu

Trong chương này, chúng ta sẽ nghiên cứu bảng băm. Bảng băm là cấu trúc dữ liệu được sử dụng để cài đặt KDL từ điển. Nhớ lại rằng, KDL từ điển là một tập các đối tượng dữ liệu được xem xét đến chỉ với ba phép toán tìm kiếm, xen vào và loại bỏ. Đương nhiên là chúng ta có thể cài đặt từ điển bởi danh sách, hoặc bởi cây tìm kiếm nhị phân. Tuy nhiên bảng băm là một trong các phương tiện hiệu quả nhất để cài đặt từ điển.

Kiến thức cơ bản cần thiết

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

- Cấu trúc mảng, danh sách
- Các cấu trúc điều khiển, lệnh vòng lặp.
- Lập trình hàm, thủ tục, cách gọi hàm.

Nội dung

Trong chương này, chúng ta sẽ đề cập tới các vấn đề sau đây:

- Phương pháp băm và hàm băm.
- Các chiến lược giải quyết sự va chạm.
- Cài đặt KDL từ điển bởi bảng băm.

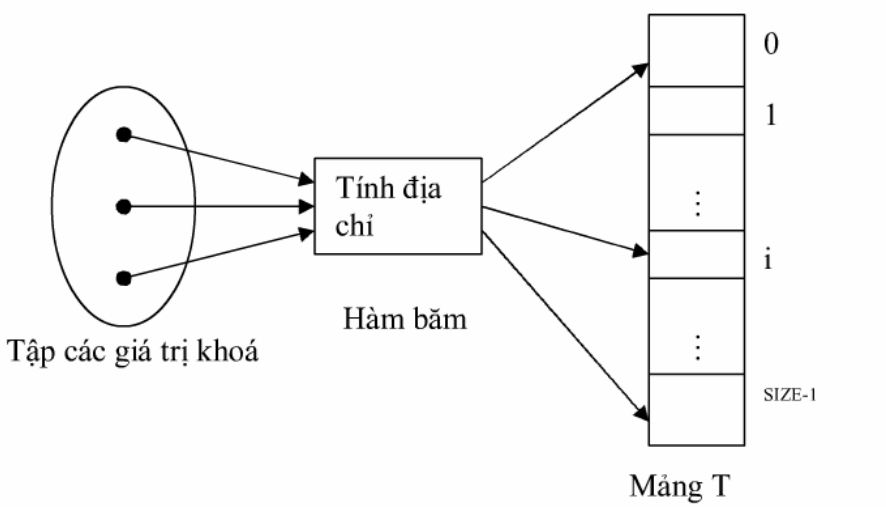
I. Phương pháp băm

Vấn đề được đặt ra là, chúng ta có một tập dữ liệu, chúng ta cần đưa ra một cấu trục dữ liệu (CTDL) cài đặt tập dữ liệu này sao cho các phép toán tìm kiếm, xen, loại được thực hiện hiệu quả. Trong các chương trước, chúng ta đã trình bày các phương pháp cài đặt KDL tập động (từ điển là trường hợp riêng của tập động khi mà chúng ta chỉ quan tâm tới ba phép toán tìm kiếm, xen, loại). Sau đây chúng ta trình bày một kỹ thuật mới để lưu giữ một tập dữ liệu, đó là phương pháp băm.

Nếu như các giá trị khoá của các dữ liệu là số nguyên không âm và nằm trong khoảng [0..SIZE-1], chúng ta có thể sử dụng một mảng data có cỡ SIZE để lưu tập dữ liệu đó. Dữ liệu có khoá là k sẽ được lưu trong thành phần data[k] của mảng. Bởi vì mảng cho phép ta truy cập trực tiếp tới từng thành phần của mảng theo chỉ số, do đó các phép toán tìm kiếm, xen, loại được thực hiện trong thời gian O(1). Song đáng tiếc là, khoá có thể không phải là số nguyên, thông thường khoá còn có thể là số thực, là ký tự hoặc xâu ký tự. Ngay cả khoá là số nguyên, thì các giá trị khoá nói chung không chạy trong khoảng [0..SIZE-1].

Trong trường hợp tổng quát, khi khoá không phải là các số nguyên trong khoảng [0..SIZE-1], chúng ta cũng mong muốn lưu tập dữ liệu bởi mảng, để lợi dụng tính ưu việt cho phép truy cập trực tiếp của mảng. Giả sử chúng ta muốn lưu tập dữ liệu trong mảng T với cỡ là SIZE. Để làm được điều đó, với mỗi dữ liệu chúng ta cần định vị

được vị trí trong mảng tại đó dữ liệu được lưu giữ. Nếu chúng ta đưa ra được cách tính chỉ số mảng tại đó lưu dữ liệu thì chúng ta có thể lưu tập dữ liệu trong mảng theo sơ đồ Hình III.1.



Hình III.1. Lược đồ phương pháp băm.

Trong lược đồ Hình III.1, khi cho một dữ liệu có khoá là k, nếu tính địa chỉ theo k ta thu được chỉ số i, 0 <= i <= SIZE-1, thì dữ liệu sẽ được lưu trong thành phần mảng T[i].

Một hàm ứng với mỗi giá trị khoá của dữ liệu với một địa chỉ (chỉ số) của dữ liệu trong mảng được gọi là hàm băm (hash function). Phương pháp lưu tập dữ liệu theo lược đồ trên được gọi là phương pháp băm (hashing). Trong lược đồ II.1, mảng T được gọi là bảng băm (hash table).

Như vậy, hàm băm là một ánh xạ h từ tập các giá trị khoá của dữ liệu vào tập các số nguyên {0,1,…, SIZE-1}, trong đó SIZE là cỡ của mảng dùng để lưu tập dữ liệu, tức là:

$h : K \rightarrow \{0,1,\ldots,SIZE-1\}$

với K là tập các giá trị khoá. Cho một dữ liệu có khoá là k, thì h(k) được gọi là giá trị băm của khoá k, và dữ liệu được lưu trong T[h(k)].

Nếu hàm băm cho phép ứng các giá trị khoá khác nhau với các chỉ số khác nhau, tức là nếu $k_1 \neq k_2$ thì $h(k_1) \neq h(k_2)$, và việc tính chỉ số h(k) ứng với mỗi khoá k chỉ đòi hỏi thời gian hằng, thì các phép toán tìm kiếm, xen, loại cũng chỉ cần thời gian O(1). Tuy nhiên, trong thực tế một hàm băm có thể ánh xạ hai hay nhiều giá trị khoá tới cùng một chỉ số nào đó. Điều đó có nghĩa là chúng ta phải lưu các dữ liệu đó trong cùng một thành phần mảng, mà mỗi thành phần mảng chỉ cho phép lưu một dữ liệu ! Hiện tượng này được gọi là sự va chạm (collision). Vấn đề đặt ra là, giải quyết sự va chạm như thế nào? Chẳng hạn, giả sử dữ liệu d_1 với khoá k_1 đã được lưu trong T[i], $i = h(k_1)$; bây giờ chúng ta cần xen vào dữ liệu d_2 với khoá k_2, nếu $h(k_2) = i$ thì dữ liệu d_2 cần được đặt vào vị trí nào trong mảng?

Như vậy, một hàm băm như thế nào thì được xem là tốt. Từ những điều đã nêu trên, chúng ta đưa ra các tiêu chuẩn để thiết kế một hàm băm tốt như sau:

1. Tính được dễ dàng và nhanh địa chỉ ứng với mỗi khoá.
2. Đảm bảo ít xảy ra va chạm.

II. Các hàm băm

Trong các hàm băm được đưa ra dưới đây, chúng ta sẽ ký hiệu k là một giá trị khoá bất kỳ và SIZE là cỡ của bảng băm. Trước hết chúng ta sẽ xét trường hợp các giá trị khoá là các số nguyên không âm. Nếu không phải là trường hợp này (chẳng hạn, khi các giá trị khoá là các xâu ký tự), chúng ta chỉ cần chuyển đổi các giá trị khoá thành các số nguyên không âm, sau đó băm chúng bằng một phương pháp cho trường hợp khoá là số nguyên.

Có nhiều phương pháp thiết kế hàm băm đã được đề xuất, nhưng được sử dụng nhiều nhất trong thực tế là các phương pháp được trình bày sau đây:

1. Phương pháp chia

Phương pháp này đơn giản là lấy phần dư của phép chia khoá k cho cỡ bảng băm SIZE làm giá trị băm: h(k) = k mod SIZE

Bằng cách này, giá trị băm h(k) là một trong các số 0,1,…, SIZE-1. Hàm băm này được cài đặt trong C++ như sau:

```
unsigned int hash(int k, int SIZE)
{
        return k % SIZE;
}
```

Trong phương pháp này, để băm một khoá k chỉ cần một phép chia, nhưng hạn chế cơ bản của phương pháp này là để hạn chế xảy ra va chạm, chúng ta cần phải biết cách lựa chọn cỡ của bảng băm. Các phân tích lý thuyết đã chỉ ra rằng, để hạn chế va chạm, khi sử dụng phương pháp băm này chúng ta nên lựa chọn SIZE là số nguyên tố, tốt hơn là số nguyên tố có dạng đặc biệt, chẳng hạn có dạng 4k+3. Ví dụ, có thể chọn SIZE = 811, vì 811 là số nguyên tố và 811 = 4 . 202 + 3.

2. Phương pháp nhân

Phương pháp chia có ưu điểm là rất đơn giản và dễ dàng tính được giá trị băm, song đối với sự va chạm nó lại rất nhạy cảm với cỡ của bảng băm. Để hạn chế sự va chạm, chúng ta có thể sử dụng phương pháp nhân, phương pháp này có ưu điểm là ít phụ thuộc vào cỡ của bảng băm.

Phương pháp nhân tính giá trị băm của khoá k như sau. Đầu tiên, ta tính tích của khoá k với một hằng số thực α, $0 < \alpha < 1$. Sau đó lấy phần thập phân của tích αk nhân với SIZE, phần nguyên của tích này được lấy làm giá trị băm của khoá k. Tức là:

$$h(k) = \lfloor (\alpha k - \lfloor \alpha k \rfloor) \, . \, SIZE \rfloor$$

(Ký hiệu $\lfloor x \rfloor$ chỉ phần nguyên của số thực x, tức là số nguyên lớn nhất <=x, chẳng hạn $\lfloor 3 \rfloor = 3, \lfloor 3.407 \rfloor = 3$)

Chú ý rằng, phần thập phân của tích α k, tức là α k - $\lfloor \alpha k \rfloor$, là số thực dương nhỏ hơn 1. Do đó tích của phần thập phân với SIZE là số dương nhỏ hơn SIZE. Từ đó, giá trị băm h(k) là một trong các số nguyên 0,1,…, SIZE- 1.

Để có thể phân phối đều các giá trị khoá vào các vị trí trong bảng băm, trong thực tế người ta thường chọn hằng số α như sau:

$$\alpha = \Phi^{-1} \approx 0{,}61803399$$

Chẳng hạn, nếu cỡ bảng băm là SIZE = 1024 và hằng số α được chọn như trên, thì với k = 1849970, ta có:

$$h(k) = \lfloor (1024.(\alpha.1849970 - \lfloor \alpha.1849970 \rfloor)) \rfloor = 348.$$

3. Hàm băm cho các giá trị khoá là xâu ký tự

Để băm các xâu ký tự, trước hết chúng ta chuyển đổi các xâu ký tự thành các số nguyên. Các ký tự trong bảng mã ASCII gồm 128 ký tự được đánh số từ 0 đến 127, đo đó một xâu ký tự có thể xem như một số trong hệ đếm cơ số 128. Áp dụng phương pháp chuyển đổi một số trong hệ đếm bất kỳ sang một số trong hệ đếm cơ số 10, chúng ta sẽ chuyển đổi được một xâu ký tựthành một số nguyên. Chẳng hạn, xâu "NOTE" được chuyển thành một số nguyên như sau:

"NOTE" → 'N'.128^3 + 'O'.128^2 + 'T'.128 + 'E' =

$$78.128^3 + 79.128^2 + 84.128 + 69$$

Vấn đề nảy sinh với cách chuyển đổi này là, chúng ta cần tính các luỹ thừa của 128, với các xâu ký tự tương đối dài, kết quả nhận được sẽ là một số nguyên cực lớn vượt quá khả năng biểu diễn của máy tính.

Trong thực tế, thông thường một xâu ký tự được tạo thành từ 26 chữ cái và 10 chữ số, và một vài ký tự khác. Do đó chúng ta thay 128 bởi 37 và tính số nguyên ứng với xâu ký tự theo luật Horner. Chẳng hạn, số nguyên ứng với xâu ký tự "NOTE" được tính như sau:

"NOTE" → $78.37^3 + 79.37^2 + 84.37 + 69$ =

$$((78.37 + 79).37 + 84).37 + 69$$

Sau khi chuyển đổi xâu ký tự thành số nguyênbằng phương pháp trên, chúng ta sẽ áp dụng phương pháp chia để tính giá trị băm. Hàm băm các xâu ký tự được cài đặt như sau:

```
unsigned   int   hash(const string &k, int SIZE)
{
    unsigned  int  value = 0;
    for (int i=0; i< k.length(); i++)
    value = 37 * value + k[i];
    return    value % SIZE;
}
```

III. Các phương pháp giải quyết va chạm

Trong mục II.2 chúng ta đã trình bày các phương pháp thiết kế hàm băm nhằm hạn chế xẩy ra va chạm. Tuy nhiên trong các ứng dụng, sự va chạm là không tránh khỏi. Chúng ta sẽ thấy rằng, cách giải quyết va chạm ảnh hưởng trực tiếp đến hiệu quả của các phép toán từ điển trên bảng băm. Trong mục này chúng ta sẽ trình bày hai phương pháp giải quyết va chạm. Trong phương pháp thứ nhất, mỗi khi xảy ra va chạm, chúng ta tiến hành thăm dò để tìm một vị trí còn trống trong bảng và đặt dữ liệu mới vào đó. Một phương pháp khác là, chúng ta tạo ra một cấu trúc dữ liệu lưu giữ tất cả các dữ liệu được băm vào cùng một vị trí trong bảng và "gắn" cấu trúc dữ liệu này vào vị trí đó trong bảng.

1. Phương pháp định địa chỉ mở

Trong phương pháp này, các dữ liệu được lưu trong các thành phần của mảng, mỗi thành phần chỉ chứa được một dữ liệu. Vì thế, mỗi khi cần xen một dữ liệu mới với khoá k vào mảng, nhưng tại vị trí h(k) đã chứa dữ liệu, chúng ta sẽ tiến hành thăm dò một số vị trí khác trong mảng để tìm ra một vị trí còn trống và đặt dữ liệu mới vào vị trí đó. Phương pháp tiến hành thăm dò để phát hiện ra vị trí trống được gọi là phương pháp định địa chỉ mở (open addressing).

Giả sử vị trí mà hàm băm xác định ứng với khoá k là i, i=h(k). Từ vị trí này chúng ta lần lượt xem xét các vị trí i_0 , i_1 , i_2 ,…, i_m ,…

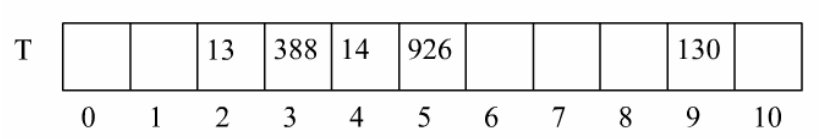
Trong đó $i_0 = i$, i_m(m=0,1,2,…) là vị trí thăm dò ở lần thứ m. Dãy các vị trí này sẽ được gọi là dãy thăm dò. Vấn đề đặt ra là, xác định dãy thăm dò như thế nào? Sau đây chúng ta sẽ trình bày một số phương pháp thăm dò và phân tích ưu khuyết điểm của mỗi phương pháp.

Thăm dò tuyến tính.

Đây là phương pháp thăm dò đơn giản và dễ cài đặt nhất. Với khoá k, giả sử vị trí được xác định bởi hàm băm là i=h(k), khi đó dãy thăm dò là i , i+1, i+2 , …

Như vậy thăm dò tuyến tính có nghĩa là chúng ta xem xét các vị trí tiếp liền nhau kể từ vị trí ban đầu được xác định bởi hàm băm. Khi cần xen vào một dữ liệu mới với khoá k, nếu vị trí i = h(k) đã bị chiếm thì ta tìmđến các vị trí đi liền sau đó, gặp vị trí còn trống thì đặt dữ liệu mới vào đó.

Ví dụ. Giả sử cỡ của mảng SIZE = 11. Ban đầu mảng T rỗng, và ta cần xen lần lượt các dữ liệu với khoá là 388, 130, 13, 14, 926 vào mảng. Băm khoá 388, h(388) = 3, vì vậy 388 được đặt vào T[3]; h(130) = 9, đặt 130 vào T[9]; h(13) = 2, đặt 13 trong T[2]. Xét tiếp dữ liệu với khoá 14, h(14) = 3, xẩy ra va chạm (vì T[3] đã bị chiếm bởi 388), ta tìm đến vị trí tiếp theo là 4, vị trí này trống và 14 được đặt vào T[4]. Tương tự, khi xen vào 926 cũng xảy ra va chạm, h(926) = 2, tìm đến các vị trí tiếp theo 3, 4, 5 và 92 được đặt vào T[5]. Kết quả là chúng ta nhận được mảng T như trong Hình III.2.



Hình III.2. Bảng băm sau khi xen vào các dữ liệu 38, 130, 13, 14 và 926

Bây giờ chúng ta xét xem, nếu lưu tập dữ liệu trong mảng bằng phương pháp định địa chỉ mở thì các phép toán tìm kiếm, xen, loại được tiến hành như thế nào. Các kỹ thuật tìm kiếm, xen, loại được trình bày dưới đây có thể sử dụng cho bất kỳ phương pháp thăm dò nào. Trước hết cần lưu ý rằng, để tìm, xen, loại chúng ta phải sử dụng cùng một phương pháp thăm dò, chẳng hạn thăm dò tuyến tính. Giả sử chúng ta cần tìm dữ liệu với khoá là k. Đầu tiên cần băm khoá k, giả sử h(k)=i. Nếu trong bảng ta chưa một lần nào thực hiện phép toán loại, thì chúng ta xem xét các dữ liệu chứa trong mảng tại vị trí i và các vị trí tiếp theo trong dãy thăm dò, chúng ta sẽ pháthiện ra dữ liệu cần tìm tại một vị trí nào đó trong dãy thăm dò, hoặc nếu gặp một vị trí trống trong dãy thăm dò thì có thể dừng lại và kết luận dữ liệu cần tìm không có trong mảng. Chẳng hạn chúng ta muốn tìm xem mảng trong Hình III.2 có chứa dữ liệu với khoá là 47? Bởi vì h(47) = 3, và dữ liệu được lưu theo phương pháp thăm dò tuyến tính, nên chúng ta lần lượt xem xét các vị trí 3, 4, 5. Các vị trí này đều chứa dữ liệu khác với 47. Đến vị trí 6, mảng trống. Vậy ta kết luận 47 không có trong mảng.

Để loại dữ liệu với khoá k, trước hết chúng ta cần áp dụng thủ tục tìm kiếm đã trình bày ở trên để định vị dữ liệu ở trong mảng. Giả sử dữ liệu được lưu trong mảng tại vị trí p. Loại dữ liệu ở vị trí p bằng cách nào? Nếu đặt vị trí p là vị trí trống, thì khi tìm kiếm nếu thăm dò gặp vị trí trống ta không thể dừng và đưa ra kết luận dữ liệu không có trong mảng. Chẳng hạn, trong mảng Hình III.2, ta loại dữ liệu 388 bằng cách xem vị trí 3 là trống, sau đó ta tìm dữ liệu 926, vì h (926) = 2 và T[2] không chứa 926, tìmđến vị trí 3 là trống, nhưng ta không thể kết luận 926 không có trong mảng. Thực tế 926 ở vị trí 5, vì lúc đưa 926 vào mảng các vị trí 2, 3, 4 đã bị chiếm. Vì vậy để đảm bảo thủ tục tìm kiếm đã trình bày ở trên vẫn còn đúng cho trường hợp đã thực hiện phép toán loại, khi loại dữ liệu ở vị trí p chúng ta đặt vị trí p là vị trí đã loại bỏ. Như vậy, chúng ta quan niệm mỗi vị trí i trong mảng (0 <= i <= SIZE-1) có thể là vị trí trống (EMPTY), vị trí đã loại bỏ (DELETED), hoặc vị trí chứa dữ liệu (ACTIVE). Đương nhiên là khi xen vào dữ liệu mới, chúng ta có thể đặt nó vào vị trí đã loại bỏ.

Việc xen vào mảng một dữ liệu mới được tiến hành bằng cách lần lượt xem xét các vị trí trong dãy thăm dò ứng với mỗi khoá của dữ liệu, khi gặp một vị trí trống hoặc vị trí đã được loại bỏ thì đặt dữ liệu vào đó.

Sau đây là hàm thăm dò tuyến tính

```
int     Probing (int i, int m, int SIZE)
// SIZE là cỡ của mảng
// i là vị trí ban đầu được xác định bởi băm khoá k, i = h(k)
// hàm trả về vị trí thăm dò ở lần thứ  m= 0, 1, 2,...
{
```

```
        return    (i+ m)  %   SIZE;
    }
```

Phương pháp thăm dò tuyến tính có ưu điểm là cho phép ta xem xét tất cả các vị trí trong mảng, và do đó phép toán xen vào luôn luôn thực hiện được, trừ khi mảng đầy. Song nhược điểm của phương pháp này là các dữ liệu tập trung thành từng đoạn, trong quá trình xen các dữ liệu mới vào, các đoạn có thể gộp thành đoạn dài hơn. Điều đó làm cho các phép toán kém hiệu quả, chẳng hạn nếu i = h(k) ở đầu một đoạn, để tìm dữ liệu với khoá k chúng ta cần xem xét cả một đoạn dài.

Thăm dò bình phương

Để khắc phục tình trạng dữ liệu tích tụ thành từng cụm trong phương pháp thăm dò tuyến tính, chúng ta không thăm dò các vị trí kế tiếp liền nhau, mà thăm dò bỏ chỗ theo một quy luật nào đó.

Trong thăm dò bình phương, nếu vị trí ứng với khoá k là i = h(k), thì dãy thăm dò là

$$i , i+ 1^2 , i+ 2^2 ,\dots , i+ m^2 ,\dots$$

Ví dụ. Nếu cỡ của mảng SIZE = 11, và i = h(k) = 3, thì thăm dò bình phương cho phép ta tìmđến các địa chỉ 3, 4, 7, 1, 8 và 6.

Phương pháp thăm dò bình phương tránh được sự tích tụ dữ liệu thành từng đoạn và tránh được sự tìm kiếm tuần tự trong các đoạn. Tuy nhiên nhược điểm của nó là không cho phép ta tìm đến tất cả các vị trí trong mảng, chẳng hạn trong ví dụ trên, trong số 11 vị trí từ 0, 1, 2, …, 10, ta chỉ tìm đến các vị trí 3, 4, 7, 1, 8 và 6. Hậu quả của điều đó là, phép toán xen vào có thể không thực hiện được, mặc dầu trong mảng vẫn còn các vị trí không chứa dữ.

Băm kép

Phương pháp băm kép (double hashing) có ưu điểm như thăm dò bình phương là hạn chế được sự tích tụ dữ liệu thành cụm; ngoài ra nếu chúng ta chọn cỡ của mảng là số nguyên tố, thì băm kép còn cho phép ta thăm dò tới tất cả các vị trí trong mảng.

Trong thăm dò tuyến tính hoặc thăm dò bình phương, các vị trí thăm dò cách vị trí xuất phát một khoảng cách hoàn toàn xác định trước và các khoảng cách này không phụ thuộc vào khoá. Trong băm kép, chúng ta sử dụng hai hàm băm h_1 và h_2:

- Hàm băm h_1 đóng vai trò như hàm băm h trong các phương pháp trước, nó xác định vị trí thăm dò đầu tiên.
- Hàm băm h_2 xác định bước thăm dò.

Điều đó có nghĩa là, ứng với mỗi khoá k, dãy thăm dò là:

$$h_1(k) + m\, h_2(k), \text{ với } m= 0, 1, 2, \dots$$

Bởi vì $h_2(k)$ là bước thăm dò, nên hàm băm h_2 phải thoả mãn điều kiện $h_2(k) \neq 0$ với mọi k.

Có thể chứng minh được rằng, nếu cỡ của mảng và bước thăm dò $h_2(k)$ nguyên tố cùng nhau thì phương pháp băm kép cho phép ta tìm đến tất cả các vị trí trong mảng. Khẳng định trên sẽ đúng nếu chúng ta lựa chọn cỡ của mảng là số nguyên tố.

Ví dụ. Giả sử SIZE = 11, và các hàm băm được xác định như sau:

$h_1(k) = k \% 11$

$h_2(k) = 1 + (k \% 7)$

với k = 58, thì bước thăm dò là $h_2(58) = 1 + 2 = 3$, do đó dãy thăm dò là: $h_1(58) = 3, 6, 9, 1, 4, 7, 10, 2, 5, 8, 0$. còn với k = 36, thì bước thăm dò là $h_2(36) = 1 + 1 = 2$, và dãy thăm dò là 3, 5, 7, 9, 0, 2, 4, 6, 8, 10.

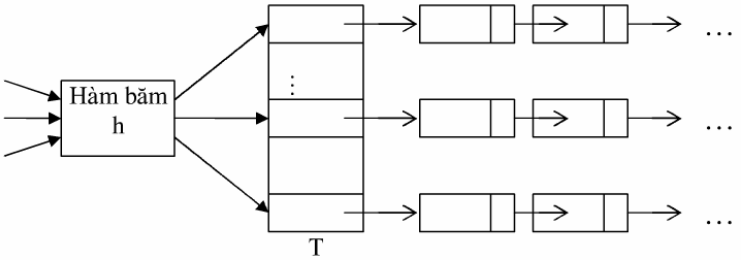
Trong các ứng dụng, chúng ta có thể chọn cỡ mảng SIZE là số nguyên tố và chọn M là số nguyên tố, M < SIZE, rồi sử dụng các hàm băm

$h_1(k) = k \% SIZE$

$h_2(k) = 1 + (k \% M)$

2. Phương pháp tạo dây chuyền

Một cách tiếp cận khác để giải quyết sự va chạm là chúng ta tạo một cấu trúc dữ liệu để lưu tất cả các dữ liệu được băm vào cùng một vị trí trong mảng. Cấu trúc dữ liệu thích hợp nhất là danh sách liên kết (dây chuyền). Khi đó mỗi thành phần trong bảng băm T[i], với i = 0, 1, …, SIZE – 1, sẽ chứa con trỏ trỏ tới đầu một DSLK. Cách giải quyết va chạm như trên được gọi là phương pháp tạo dây chuyền (separated chaining). Lược đồ lưu tập dữ liệu trong bảng băm sử dụng phương pháp tạo dây chuyền được mô tả trong Hình III.3.



Hình III.3. Phương pháp tạo dây chuyền

Ưu điểm của phương pháp giải quyết va chạm này là số dữ liệu được lưu không phụ thuộc vào cỡ của mảng, nó chỉ hạn chế bởi bộ nhớ cấp phát động cho các dây chuyền.

Bây giờ chúng ta xét xem các phép toán từ điển (tìm kiếm, xen, loại) được thực hiện như thế nào. Các phép toán được thực hiện rất dễ dàng, để xen vào bảng băm dữ liệu khoá k, chúng ta chỉ cần xen dữ liệu này vào đầu DSLK được trỏ tới bởi con trỏ T[h(k)]. Phép toán xen vào chỉ đòi hỏi thời gian O(1), nếu thời gian tính giá trị băm h(k) là O(1). Việc tìm kiếm hoặc loại bỏ một dữ liệu với khoá k được quy về tìm kiếm hoặc loại bỏ trên DSLK T[h(k)]. Thời gian tìm kiếm hoặc loại bỏ đương nhiên là phụ thuộc vào độ dài của DSLK.

Chúng ta có nhận xét rằng, dù giải quyết va chạm bằng cách thăm dò, hay giải quyết va chạm bằng cách tạo dây chuyền, thì bảng băm đều không thuận tiện cho sự thực

hiện các phép toán tập động khác, chẳng hạn phép toán Min (tìm dữ liệu có khoá nhỏ nhất), phép toán DeleteMin (loại dữ liệu có khoá nhỏ nhất), hoặc phép duyệt dữ liệu.

Sau này chúng ta sẽ gọi bảng băm với giải quyết va chạm bằng phương pháp định địa chỉ mở là bảng băm địa chỉ mở, còn bảng băm giải quyết va chạm bằng cách tạo dây chuyền là bảng băm dây chuyền.

IV. Cài đặt bảng băm địa chỉ mở

Trong mục này chúng ta sẽ nghiên cứu sự cài đặt KDL từ điển bởi bảng băm địa chỉ mở. Chúng ta sẽ giả thiết rằng, các dữ liệu trong từ điển có kiểu Item nào đó, và chúng chứa một trường dùng làm khoá tìm kiếm (trường key), các giá trị khoá có kiểu int. Ngoài ra để đơn giản cho viết ta giả thiết rằng, có thể truy cập trực tiếp trường key. Như đã thảo luận trong mục III.1, trong bảng băm T, mỗi thành phần T[i], 0 <= i <= SIZE -1, sẽ chứa hai biến: biến data để lưu dữ liệu và biến state để lưu trạng thái của vị trí i, trạng thái của vị trí i có thể là rỗng (EMPTY), có thể chứa dữ liệu (ACTIVE), hoặc có thể đã loại bỏ (DELETED). Chúng ta sẽ cài đặt KDL bởi lớp OpenHash phụ thuộc tham biến kiểu Item, lớp này sử dụng một hàm băm Hash và một hàm thăm dò Probing đã được cung cấp. Lớp OpenHash được khai báo như sau:

```
const   int   SIZE = 811;
enum   stateType {ACTIVE, EMPTY, DELETED};
struct     Entry
{
    int     data;
    stateType    state;
}
Entry T[SIZE];
OpenHash(); // khởi tạo bảng băm rỗng.
bool     Search(int   k, Item & I)  const;
        // Tìm dữ liệu có khoá là k.
        // Hàm trả về true (false) nếu tìm thấy (không tìm thấy).
        // Nếu tìm kiếm thành công, biếnI ghi lại dữ liệu cần tìm.
void     Insert(const Item &  object, bool &  Suc)
        // Xen vào dữ liệu object. biến Suc nhận giá trị true
        // nếu phép xen thành công, và false nếu thất bại.
void     Delete(int   k);
        // Loại khỏi bảng băm dữ liệu có khoá k.
bool   Find(int k, int & index, int & index1)   const;
        // Hàm thực hiện thăm dò tìm dữ liệu có khoá k.
```

```
        // Nếu thành công, hàm trả về true và biến index ghi lại chỉ
        // số tại đó chứa dữ liệu.
        // Nếu thất bại, hàm trả về false và biến index1 ghi lại
        // chỉ số ở trạng thái EMPTY hoặc DELETED nếu thăm dò
        // phát hiện ra.
```

Để khởi tạo bảng băm rỗng như sau:

```
for ( int i= 0 ; i < SIZE ; i++ )
        T[i].state = EMPTY;
```

Chú ý rằng, các phép toán tìm kiếm, xen, loại đều cần phải thực hiện thăm dò để phát hiện ra dữ liệu cần tìm hoặc để phát hiện ra vị trí rỗng (hoặc bị trí đã loại bỏ) để đưa vào dữ liệu mới. Sử dụng hàm Find ta dễ dàng cài đặt được các hàm Search, Insert và Delete. Trước hết chúng ta cài đặt hàm Find. Trong hàm Find khi mà quá trình thăm dò phát hiện ra vị trí rỗng thì có nghĩa là bảng không chứa dữ liệu cần tìm, song trước khi đạt tới vị trí rỗng có thể ta đã phát hiện ra các vị trí đã loại bỏ, biến index1 sẽ ghi lại vị trí đã loại bỏ đầu tiên đã phát hiện ra . Còn nếu phát hiện ra vị trí rỗng, nhưng trước đó ta không gặp vị trí đã loại bỏ nào, thì biến index1 sẽ ghi lại vị trí rỗng. Hàm Find được cài đặt như sau:

```
bool Find(int k, int & index, int & index1)
{
    int i = Hash(k);
    index = 0;
    index1 = i;
    for (int m= 0 ; m< SIZE ; m++)
    {
        int   n = Probing(i,m); // vị trí thăm dò ở lần thứ m.
        if (T[n].state = = ACTIVE && T[n].data= = k )
        {
            index = n;
            return   true;
        }
        else if (T[n].state = = EMPTY)
        {
            if (T[index1].state != DELETED)
            index1 = n;
```

```
            return   false;
        }
        else if (T[n].state = = DELETED && T[index1].state!= DELETED)
                index1 = n;
    }
    return   false; // Dừng thăm dò mà vẫn không tìm ra dữ liệu
    //  và cũng không phát hiện ra vị trí rỗng.
}
```

Sử dụng hàm Find, các hàm tìm kiếm, xen, loại được cài đặt như sau:

```
bool   Search(int k)
{
    int   ind, ind1;
    if (Find(k,ind,ind1))
    {
        return    true;
    }
    else
    {
        return   false;
    }
}
void   Insert(int & object, bool &  Suc)
{
    int    ind, ind1;
    if (!Find(object, ind, ind1))
        if (T[ind1].state = = DELETED || T[ind1].state = = EMPTY)
        {
            T[ind1].data  =  object;
            T[ind1].state  = ACTIVE;
            Suc  =  true;
        }
        else Suc  =  false;
```

```
}
void   Delete(int   k)
{
    int   ind, ind1;
    if (Find(k, ind, ind1))
    T[ind].state = DELETED;
}
```

Trên đây chúng ta đã cài đặt bảng băm địa chỉ mở bởi mảng có cỡ cố định. Hạn chế của cách này là, phép toán Insert có thể không thực hiện được do mảng đầy hoặc có thể mảng không đầy nhưng thăm dò không phát hiện ra vị trí rỗng hoặc vị trí đã loại bỏ để đặt dữ liệu vào. Câu hỏi đặt ra là, chúng ta có thể cài đặt bởi mảng động như chúng ta đã làm khi cài đặt KDL tập động. Câu trả lời là có, tuy nhiên cài đặt bảng băm bởi mảng động sẽ phức tạp hơn, vì các lý do sau:

- Cỡ của mảng cần là số nguyên tố, do đó chúng ta cần tìm số nguyên tố tiếp theo SIZE làm cỡ của mảng mới.
- Hàm băm phụ thuộc vào cỡ của mảng, chúng ta không thể sao chép một cách đơn giản mảng cũ sang mảng mới như chúng ta đã làm trước đây, mà cần phải sử dụng hàm Insert để xen từng dữ liệu của bảng cũ sang bảng mới

V. Cài đặt bảng băm dây chuyền

Trong mục này chúng ta sẽ cài đặt KDL từ điển bởi bảng băm dây chuyền. Lớp ChainHash phụ thuộc tham biến kiểu Item với các giả thiết như trong mục IV. Lớp này được định nghĩa như sau:

```
struct     Cell
{
    Item    data;
    Cell*   next;
};  // Cấu trúc tế bào trong dây chuyền.
Cell*   T[SIZE];// Mảng các con trỏ trỏ  đầu các dây chuyền
// Các phép toán từ điển:
bool   Search(int k, Item & I)  const;
void    Insert(const Item & object, bool & Suc);
void    Delete(int   k);
```

Để khởi tạo ra bảng băm rỗng, chúng ta chỉ cần đặt các thành phần trong mảng T là con trỏ NULL.

Hàm kiến tạo mặc định như sau:

```
for ( int i= 0 ; i< SIZE ; i++ )
        T[i] = NULL;
```

Các hàm tìm kiếm, xen, loại được cài đặt rất đơn giản, sau khi băm chúng ta chỉ cần áp dụng các kỹ thuật tìm kiếm, xen, loại trên các DSLK. Các hàm Search, Insert và Delete được xác định dưới đây:

```
bool Search(int k, Item & I)
{
    int  i = Hash(k);
    Cell*  P = T[i];
    while (P ! = NULL)
        if (P→data.key = = k)
        {
            I = P→data;
            return   true;
        }
        else P= P→next;
    return    false;
}
void   Insert(const Item &  object, bool &  Suc)
{
    int   i = Hash(k);
    Cell*   P = new Cell;
    If (P != NULL)
    {
        P→data = object;
        P→next = T[i];
        T[i] = P;  //Xen vào đầu dây chuyền.
        Suc = true;
    }
    else    Suc  = false;
}
void   Delete(int  k)
{
```

```
int   i = Hash(k);
Cell*   P;
If (T[i] != NULL)
    If (T[i] →data.key = = k)
    {
        P = T[i];
        T[i] = T[i] →next;
        delete   P;
    }
    else
    {
        P = T[i];
        Cell*  Q = P →next;
        while (Q != NULL)
        if (Q →data.key = = k)
        {
            P →next = Q →next;
            delete   Q;
            Q = NULL;
        }
        else
        {
            P  =  Q;
            Q  =  Q →next;
        }
    }
}
```

Ưu điểm lớn nhất của bảng băm dây chuyền là, phép toán Insert luôn luôn được thực hiện, chỉ trừ khi bộ nhớ để cấp phát động đã cạn kiệt. Ngoài ra, các phép toán tìm kiếm, xen, loại, trên bảng băm dây chuyền cũng rất đơn giản. Tuy nhiên, phương pháp này tiêu tốn bộ nhớ giành cho các con trỏ trong các dây chuyền.

VI. Hiệu quả của các phương pháp băm

Trong mục này, chúng ta sẽ phân tích thời gian thực hiện các phép toán từ điển (tìm kiếm, xen, loại) khi sử dụng phương pháp băm. Trong trường hợp xấu nhất, khi mà hàm băm băm tất cả các giá trị khoá vào cùng một chỉ số mảng để tìm kiếm chẳng hạn, chúng ta cần xem xét từng dữ liệu giống như tìm kiếm tuần tự, vì vậy thời gian các phép toán đòi hỏi là O(N), trong đó N là số dữ liệu.

Sau đây chúng ta sẽ đánh giá thời gian trung bình cho các phép toán từ điển. Đánh giá này dựa trên giả thiết hàm băm phân phối đều các khoá vào các vị trí trong bảng băm (uniform hashing). Chúng ta sẽ sử dụng một tham số α, được gọi là mức độ đầy (load factor). Mức độ đầy α là tỷ số giữa số dữ liệu hiện có trong bảng băm và cỡ của bảng, tức là:

$$\alpha = \frac{N}{SIZE}$$

trong đó, N là số dữ liệu trong bảng. Rõ ràng là, khi α tăng thì khả năng xảy ra va chạm sẽ tăng, điều này kéo theo thời gian tìm kiếm sẽ tăng. Như vậy hiệu quả của các phép toán phụ thuộc vào mức độ đầy α. Khi cỡ mảng cố định, hiệu quả sẽ giảm nếu số dữ liệu N tăng lên. Vì vậy, trong thực hành thiết kế bảng băm, chúng ta cần đánh giá số tối đa các dữ liệu cần lưu để lựa chọn cỡ SIZE sao cho α đủ nhỏ. Mức độ đầy α không nên vượt quá 2/3.

Thời gian tìm kiếm cũng phụ thuộc sự tìm kiếm là thành công hay thất bại. Tìm kiếm thất bại đòi hỏi nhiều thời gian hơn tìm kiếm thành công, chẳng hạn trong bảng băm dây chuyền chúng ta phải xem xét toàn bộ một dây chuyền mới biết không có dữ liệu trong bảng.

D.E. Knuth (trong The art of computer programming, vol3) đã phân tích và đưa ra các công thức đánh giá hiệu quả cho từng phương pháp giải quyết va chạm như sau.

Thời gian tìm kiếm trung bình trên bảng băm địa chỉ mở sử dụng thăm dò tuyến tính. Số trung bình các lần thăm dò cho tìm kiếm xấp xỉ là:

Tìm kiếm thành công $\frac{1}{2}\left(1+\frac{1}{1-\alpha}\right)$

Tìm kiếm thất bại $\frac{1}{2}\left(1+\frac{1}{(1-\alpha)^2}\right)$

Trong đó α là mức độ đầy và $\alpha < 1$.

Ví dụ. Nếu cỡ bảng băm SIZE = 811, bảng chứa N = 649 dữ liệu, thì mức độ đầy là $\alpha = \frac{649}{811} \approx 80\%$ Khi đó, để tìm kiếm thành công một dữ liệu, trung bình chỉ đòi hỏi xem xét 3 vị trí mảng, vì

$$\frac{1}{2}\left(1+\frac{1}{1-\alpha}\right) = \frac{1}{2}\left(1+\frac{1}{1-0,8}\right) = 3$$

Thời gian tìm kiếm trung bình trên bảng băm địa chỉ mở sử dụng thăm dò bình phương (hoặc băm kép). Số trung bình các lần thăm dò cho tìm kiếm được đánh giá là

Tìm kiếm thành công $\frac{-\ln(1-\alpha)}{\alpha}$

Tìm kiếm thất bại $\frac{1}{1-\alpha}$

Phương pháp thăm dò này đòi hỏi số lần thăm dò ít hơn phương pháp thăm dò tuyến tính. Chẳng hạn, giả sử bảng đầy tới 80%, để tìm kiếm thành công trung bình chỉ đòi hỏi xem xét 2 vị trí mảng,

$$\frac{-\ln(1-\alpha)}{\alpha} = \frac{-\ln(1-0{,}8)}{0{,}8} = \frac{1{,}6}{0{,}8} = 2$$

Thời gian tìm kiếm trung bình trên bảng băm dây chuyền. Trong bảng băm dây chuyền, để xen vào một dữ liệu mới, ta chỉ cần đặt dữ liệu vào đầu một dây chuyền được định vị bởi hàm băm. Do đó, thời gian xen vào là O(1).

Để tìm kiếm (hay loại bỏ) một dữ liệu, ta cần xem xét các tế bào trong một dây chuyền. Đương nhiên là dây chuyền càng ngắn thì tìm kiếm càng nhanh. Độ dài trung bình của một dây chuyền là $\frac{N}{SIZE} = \alpha$ (với giả thiết hàm băm phân phối đều).

Khi tìm kiếm thành công, chúng ta cần biết dây chuyền có rỗng không, rồi cần xem xét trung bình là một nửa dây chuyền. Do đó, số trung bình các vị trí cần xem xét khi tìm kiếm thành công là $1+\frac{\alpha}{2}$

Nếu tìm kiếm thất bại, có nghĩa là ta đã xem xét tất cả các tế bào trong một dây chuyền nhưng không thấy dữ liệu cần tìm, do đó số trung bình các vị trí cần xem xét khi tìm kiếm thất bại là α.

Tóm lại, hiệu quả của phép toán tìm kiếm trên bảng băm dây chuyền là:

Tìm kiếm thành công $1+\frac{1}{\alpha}$

Tìm kiếm thất bại α

Mức độ đầy α	Bảng băm địa chỉ mở với thăm dò tuyến tính	Bảng băm địa chỉ mở với thăm dò bình phương	Bảng băm dây chuyền
0,5	1,50	1,39	1,25
0,6	1,75	1,53	1,30
0,7	2,17	1,72	1,35
0,8	3,00	2,01	1,40
0,9	5,50	2,56	1,45
1,0			1,50
2,0			2,00
3,0			3,00

Hình III.6. Số trung bình các vị trí cần xem xét trong tìm kiếm thành công.

Các con số trong bảng ở Hình III.6, và thực tiễn cũng chứng tỏ rằng, phương pháp băm là phương pháp rất hiệu quả để cài đặt từ điển.

Bài tập

1. Hãy cài đặt hàm băm sử dụng phương pháp nhân mục II.2.
2. Hãy cài đặt hàm thăm dò sử dụng phương pháp băm kép.
3. Giả sử cỡ của bảng băm là SIZE = s và $d_1, d_2, \ldots, d_{s-1}$ là hoán vị ngẫu nhiên của các số 1, 2, …, s-1. Dãy thăm dò ứng với khoá k được xác định như sau:

   $i_0 = i = h(k)$

   $i_m = (i + d_i) \% \text{ SIZE}, 1 \leq m \leq s-1$

   Hãy cài đặt hàm thăm dò theo phương pháp trên.
4. Cho cỡ bảng băm SIZE = 11. Từ bảng băm rỗng, sử dụng hàm băm chia lấy dư, hãyđưa lần lượt các dữ liệu với khoá:

   32 , 15 , 25 , 44 , 36 , 21

   vào bảng băm và đưa ra bảng băm kết quả trong các trường hợp sau:

   b. Bảng băm được chỉ mở với thăm dò tuyến tính.

   c. Bảng băm được chỉ mở với thăm dò bình phương.

   d. Bảng băm dây chuyền.
5. Từ các bảng băm kết quả trong bài tập 4, hãy loại bỏ dữ liệu với khoá là 44 rồi sau đó xen vào dữ liệu với khoá là 65.
6. Bảng băm chỉ cho phép thực hiện hiệu quả các phép toán tập động nào? Không thích hợp cho các phép toán tập động nào? Hãy giải thích tại sao?

7. Giả sử khoá tìm kiếm là từ tiếng Anh. Hãy đưa ra ít nhất 3 cách thiết kế hàm băm. Bình luận về các cách thiết kế đó theo các tiêu chuẩn hàm băm tốt.

Chương IV

Một số phương pháp thiết kế thuật giải cơ bản

Mục tiêu

Sau khi học xong chương này, sinh viên nắm được một số phương pháp thiết kế giả thuật cơ bản, cài đặt và vận dụng để giải một số 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:

- Các cấu trúc điều khiển, lệnh vòng lặp.
- Lập trình hàm, thủ tục, cách gọi hàm.
- Lập trình đệ qui và gọi đệ qui.

Nội dung

Trong chương này chúng ta sẽ nghiên cứu một số phương pháp thiết kế giải thuật cơ bản như sau:

- Phương pháp chia để trị
- Phương pháp quay lui
- Phương pháp tham lam

I. Phương pháp chia để trị

1. Mở đầu

Ý tưởng:

Có lẽ quan trọng và áp dụng rộng rãi nhất là kĩ thuật chia để trị. Nó phân rã bài toán kích thước n thành các bài toán nhỏ hơn mà việc tìm lời giải của chúng là cùng một cách. Lời giải của bài toán lớn được xây dựng từ lời giải của các bài toán con này.

Ta có thể nói vắn tắt ý tưởng chính của phương pháp này là: chia dữ liệu thành từng miền nhỏ, giải bài toán trên các miền đã chia rồi tổng hợp kết quả lại.

Mô hình

Nếu gọi D&C($\Re$) với $\Re$ là miền dữ liệu là hàm thể hiện cách giải bài toán theo phương pháp chia để trị thì ta có thể viết:

```
Void D&C(ℜ)
{
```

```
        If(ℜ đủ nhỏ)
                Giải bài toán
        Else
        {
                Chia ℜ thành ℜ1, .., ℜm;
                For(j=1; j<=m; j++)
                        D&C(ℜj);
                Tổng hợp kết quả;
        }
}
```

Sau đây là một số ví dụ minh họa cho phương pháp chia để trị

2. Tìm kiếm nhị phân

Phát biểu bài toán

Cho mảng n phần tử đã được sắp xếp tăng dần và một phần tử x. Tìm x có trong mảng hay không? Nếu có trả về kết quả là 1, ngược lại trả về kết quả là 0.

Ý tưởng

Chia đôi mảng, mỗi lần so sánh phần tử giữa với x, nếu phần tử giữa nhỏ hơn x thì tìm x ở nửa bên phải, ngược lại thì tìm x ở nửa bên phải.

Mô tả thuật toán

```
Input: a[1..n]
Output:
1: nếu x thuộc a
0: nếu x không thuộc a
Mô tả:
TKNP(a, x, đầu, cuối) ≡
        If(đầu > cuối)
                Return 0;
        Else
        {
                giữa = (đầu + cuối)/2;
                If(x == a[giữa])
                        Return 1;
                Else if(x > a[giữa])
```

TKNP(a, x, giữa +1, cuối);

Else TKNP(a, x, đầu, giữa-1);

}

Độ phức tạp của thuật toán

Trường hợp tốt nhất: ứng với trường hợp tìm thấy x trong lần so sánh đầu tiên. Ta có:

$T_{tốt}(n) = O(1)$

Trường hợp xấu nhất: độ phức tạp là O(lgn). Thật vậy, nếu gọi T(n) là độ phức tạp của thuật toán, thì sau khi kiểm tra điều kiện (x == a[giữa]) không thỏa và gọi đệ quy thuật toán này với dữ liệu giảm đi một nửa, thỏa mãn công thức truy hồi:

T(n) = 1+T(n/2); n>=2 và T[1]=0.

3. Bài toán Min-Max

Phát biểu bài toán

Tìm min-max trong đoạn a[l..r] của mảng a[1..n]

Ý tưởng

Tại mỗi bước chia đôi đoạn cần tìm rồi tìm min, max của từng đoạn, sau đó tổng hợp kết quả lại.

Nếu đoạn chia chỉ có một phần tử thì min = max và bằng phần tử đó.

Ví dụ minh họa:

J	1	2	3	4	5	6	7	8
a[j]	10	1	5	0	9	3	15	19

Tìm giá trị min, max trong đoạn a[2..7].

Ki hiệu:

MinMax(a,l,r,Min, Max) cho Min, Max trong đoạn a[l..r].

MinMax(a,2,7,Min,Max) cho Min=0, Max=15 trong đoạn a[2..7]

Mô tả thuật toán

Input: a[l..r] (l<=r)

Output:

Min = min(a[l]…a[r])

Max = max(a[l]..a[r])

Mô tả:

MinMax(a,l,r,Min,Max)

If(l==r)

```
{
        Min = a[l];
        Max = a[l];
}
Else
{
        MinMax(a,l,(l+r)/2,Min1,Max1)
        MinMax(a,(l+r)/2,r,Min2,Max2)
        If(Min1<Min2)
                Min = Min1;
        Else Min = Min2;
        If(Max1<Max2)
                Max = Max2;
        Else Max = Max1;
}
```

Độ phức tạp thuật toán

Gọi T(n) là số phép so sánh cần thực hiện. Khi đó ta có:

$$T(n) = \begin{cases} T(n/2) + T(n/2) + 2; n > 2 \\ 1; n = 2 \\ 0; n = 1 \end{cases}$$

Với $n = 2^k$, thì:

$$T(n) = 2 + 2T(n/2) = 2 + 2^2 + 2^2 T(n/2^2) = \ldots = 2^{k-1}T(2) + \sum_{i=1}^{k-1} 2^i =$$

$$= \sum_{i=1}^{k-1} 2^i - 2^{k-1} = 2^{k+1} - 2^{k-1} - 2 = 3n/2 - 2.$$

Vậy $T(n) \in O(n)$.

4. Thuật toán QuickSort

Phát biểu bài toán

Sắp xếp một mảng không có thứ tự thành một màng có thứ tự xác định,chẳng hạn tăng hoặc giảm.

Ý tưởng

Chọn ngẫu nhiên một phần tử x.

Duyệt dãy từ bên trái (theo chỉ số i) trong khi $a_i < x$.

Duyệt dãy từ bên phải (theo chỉ số j) trong khi $a_j > x$.

Đổi chỗ a[i] và a[j] nếu hai phía chưa vượt qua nhau, .. tiếp tục quá trình duyệt và đổi chỗ như trên trong khi hai phía còn chưa vượt qua nhau (tức là i<=j).

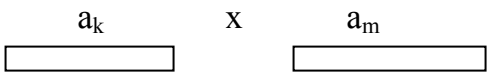
Kết quả phân hoạch dãy thành 3 phần:

a_k <=x với k = 1..j (dãy con thấp);

a_m >=x với m = i..n (dãy con cao);

a_h = x với h = j+1..i-1.

Vì vậy phương pháp này còn gọi là sắp xếp phân hoạch



Tiếp tục phân hoạch cho phần trái và phần phải cho đến khi các phân hoạch chỉ còn lại một phần tử là sắp xếp xong.

Mô tả thuật toán:

```
Input: a[l..r]
Output: a[l..r] không giảm
QuickSort(a,l,r)
{
        i=l;
        j=r;
        x= a[(l+r)/2];//chọn phần tử giữa
        do
        {
                While(a[i]<x) i++;
                While(a[j]>x)j--;
                If(i<=j)
                {
                        Đổi chỗ a[i] và a[j];
                        i++;
                        j--;
                }
        }while(i<=j)
        If(l<=j) QuickSort(a,l,j);
        If(i<=r) QuickSort(a,i,r);
```

}

Độ phức tạp thuật giải

Điều tốt nhất có thể xảy ra trong QuickSort mỗi giai đoạn phân hoạch chia mảng thành hai nửa. Điều này khiến cho số lần so sánh cần thiết của QuickSort thỏa mãn công thức sau đây:

$$T_n = 2T_{n/2} + n = nlgn.$$

$2T_{n/2}$: phí tổn sắp xếp 2 mảng con.

n: phí tổn kiểm tra mỗi phần tử

Trường hợp xấu nhất ứng cho việc chọn phần tử x lại có giá trị lớn nhất hoặc nhỏ nhất trong dãy. Giả sử phần tử lớn nhất được chọn (x), khi đó mỗi bước chia sẽ chia n phần tử thành n-1 phần tử trái và 1 phần tử phải. Kết quả cần tới n phép chia (thay cho nlgn) và như thế độ phức tạp sẽ là $T(n) = O(n^2)$.

Trong trường hợp này dãy đã có thứ tự thuận hay ngược, phần tử lớn nhất được chọn sẽ nằm ở các biên (trái hoặc phải), nên thuật toán không có hiệu quả.

Trường hợp trung bình, công thức truy hồi để tính số lần so sánh mà thuật toán cần để hoán vị ngẫu nhiên n phần tử là:

$$T(n) = (n+1) + \frac{1}{n}\sum_{1<=k<=n}(T_{k-1} + T_{n-k}); \text{ với } n>=2; C_0 = C_1 = 1;$$

Giá trị n+1 bao hàm chi phí so sánh phần tử phân hoạch với mỗi phần tử còn lại, tổng còn lại mang ý nghĩa là mỗi phần tử k có thể là phần tử phân hoạch với xác suất 1/k và sau đó còn lại các mảng con có kích thước k-1 và n-k.

$$T_n = n+1+\frac{2}{n}\sum_{k=1}^{n} T_{k-1}$$

Thực hiện liên tiếp các phép toán sau cho cả hai vế: nhân n và trừ cho $(n-1)C_{n-1}$:

$$nT_n - (n-1)T_{n-1} = n(n-1) + \frac{2}{n}\sum_{k=1}^{n} T_{k-1} - (n-1)T_{n-1}$$

$$= n(n+1) + \frac{2}{n}\sum_{k=1}^{n} T_{k-1} - (n-1)[n + \frac{2}{n-1}\sum_{k=1}^{n-1} T_{k-1}]$$

$$= n(n+1) - n(n-1) + 2\sum_{k=1}^{n} T_{k-1} - 2\sum_{k=1}^{n-1} T_{k-1}$$

Ta được: $nT_n - (n-1)T_{n-1} = n(n+1) - n(n-1)+ 2T_{n-1}$

Suy ra: $nT_n = (n+1)T_{n-1} + 2n$

Nhân cả hai vế cho n(n+1):

$T_n/(n+1) = T_{n-1}/n + 2/(n+1) = T_{n-2}/(n-1)+ 2/n + 2/(n+1)$

$= 2/(n+1) + 2/n + \ldots 2/4 + 2/3 + T_1/2 = 1/2 + 2\sum_{k=2}^{n}\frac{2}{k+1}$

$= 1/2 + 2\sum_{k=3}^{n+1}\frac{1}{k}$

$T_n/(n+1) \equiv 2\sum_{k=3}^{n+1}\frac{1}{k} \equiv 2\int_1^n \frac{1}{x}dx = 2\ln(n)$

Như vậy độ phức tạp trung bình là O(nlnn)

II. Phương pháp quay lui

1. Mở đầu

Ý tưởng

Nét đặc trưng của phương pháp quay lui là các bước hướng tới lời giải cuối cùng của bài toán đều được làm thử.

Tại mỗi bước nếu có một lựa chọn được chấp nhận thì ghi nhận lại lựa chọn này và tiến hành các bước thử tiếp theo. Còn ngược lại không có sự lựa chọn nào thích hợp thì làm lại bước trước, xóa bỏ ghi nhận và quay về chu trình thử các lựa chọn còn lại.

Hành động này được gọi là quay lui, thuật toán thể hiện phương pháp này gọi là quay lui.

Điểm quan trọng của thuật toán là phải ghi nhớ lại mỗi bước đi qua để tránh trùng lặp khi quay lui. Dễ thấy là các thông tin này cần được lưu trữ vào một ngăn xếp, nên thuật toán thể hiện một cách đệ quy.

Mô hình

Lời giải của bài toán thường được biểu diễn bằng một vector gồm n thành phần x= (x_1, ..x_n) phải thỏa các điều kiện nào đó. Để chỉ ra lời giải x ta phải xây dựng các thành phần lời giải x_i.

Tại mỗi bước i:

Đã xây dựng xong các thành phần $x_1..x_{i-1}$.

Xây dựng thành phần x_i bằng cách lần lượt thử các khả năng mà x_i có thể chọn:

- Nếu một khả năng j nào đó phù hợp với x_i thì xác định x_i theo khả năng j. Thường phải có thêm thao tác ghi nhận trạng thái mới của bài toán để hỗ trợ cho bước quay lui. Nếu i = n thì ta có được một lời giải, ngược lại thì tiến hành bước i+1 để xác định x_{i+1}.
- Nếu không có một khả năng nào chấp nhận được cho x_i thì ta lùi lại bước trước (bước i-1) để xác định lại thành phần x_{i-1}.

Để đơn giản ta có thể giả định các khả năng chọn lựa cho các x_i tại mỗi bước là như nhau, do đó ta phải có thêm thao tác kiểm tra khả năng j nào chấp nhận được cho x_i.

Mô hình của phương pháp quay lui có thể viết bằng thủ tục như sau, với n là số bước cần phải thực hiện, k là số khả năng mà x_i có thể chọn lựa

```
Try(i) ≡
For(j=1 →k)
        If(xi chấp nhận được khả năng j)
        {
                Xác định xi theo khả năng j;
                Ghi nhận trạng thái mới;
                If(i<n)
                        Try(i+1);
                Else
                        Ghi nhận nghiệm;
                Trả lại trạng thái cũ cho bài toán;
        }
```

2. Bài toán liệt kê dãy nhị phân độ dài n

Phát biểu bài toán

Liệt kê dãy có chiều dài n dưới dạng $x_1,x_2,..,x_n$ trong đó x_i thuộc {0,1}

Thiết kế thuật toán

Ta có thể sử dụng sơ đồ tìm tất cả lời giải của bài toán. Hàm Try(i) xác định x_i, trong đó x_i chỉ có một trong hai giá trị 0 hoặc 1. Hàm Try(i) có thể viết như sau:

```
Try(i)
{
        For(j=0; j<=1; j++)
        {
                x[i] = j;
                if(i<n)
                        Try(i+1);
                Else Xuất x.
        }
}
```

3. Bài toán liệt kê các hoán vị

Phát biểu bài toán

Liệt kê các hoán vị của n số nguyên dương đầu tiên

Thiết kế thuật toán

Ta biểu diễn các hoán vị dưới dạng $a_1,..,a_n$; $a_i \in \{1,..,n\}$, $a_i \neq a_j$ nếu $i \neq j$. Với mọi i, a_i chấp nhận giá trị j nếu j chưa được sử dụng và vì vậy ta cần ghi nhớ j đã được sử dụng hay chưa khi quay lui. Để làm điều này ta sử dụng một dãy các biến logic b_j với quy ước:

$$\forall j = \overline{1,n};\ b_j = \begin{cases} 1;\ \text{Nếu j chưa sử dụng} \\ 0;\ \text{Nếu j đã sử dụng} \end{cases}$$

Sau khi gán j cho a_i, ta cần ghi nhớ b_j ($b_j = 0$) và phải trả lại trạng thái cũ cho b_j ($b_j = 1$) khi thực hiện việc in xong một hoán vị.

Ta cần chú ý rằng dãy các biến b_j sẽ được khởi động bằng 1.

Thuật toán

```
Try(i)
{
        For(j=1; j<=n; j++)
        {
                If(b[j])
                {
                        a[i] = j;
                        b[j] = 0;
                        if(i<n)
                                Try(i+1);
                        Else Xuất;
                        b[j] = 1;
        }
}
```

4. Bài toán duyệt đồ thị theo chiều sâu (DFS)

Phát biểu bài toán

G = (V,U) là đơn đồ thị (có hướng hoặc vô hướng). V: tập các đỉnh của đồ thị, U là tập các cung cũa đồ thị. Với s, t là hai đỉnh của đồ thị, tìm tất cả các đường đi từ s đến t.

Ý tưởng

Thuật toán DFS tiến hành tìm kiếm trong đồ thị theo chiều sâu. Thuật toán thực hiện việc thăm tất cả các đỉnh có thể đạt được cho tới đỉnh t từ đỉnh s cho trước. Đỉnh được

thăm càng muộn sẽ càng sớm được duyệt xong (cơ chế vào sau ra trước). Nên thuật toán có thể tổ chức bằng một thủ tục đệ quy quay lui.

Mô tả thuật toán

Input: G = (V,U), s,t

Output: Tất cả các đường đi từ s đến t.(nếu có)

```
DFS(int s)
{
    For(u=1; u<=n; u++)
        If(chấp nhận được)
        {
            Ghi nhận nó;
            If(u != t)
                DFS(u);
            Else Xuất đường đi
            Bỏ việc ghi nhận;
        }
}
```

Ví dụ: Cho đồ thị có hướng với ma trận kề sau:

7

0	0	0	1	0	1	1
0	0	1	1	0	0	0
0	1	0	1	0	1	0
1	0	1	0	0	0	0
0	0	0	0	1	1	0
1	0	0	0	1	0	0
1	0	0	1	0	0	0

Kết quả:

s=1, t=4	s=2, t = 5
1→4 1→7→4	2→3→4→1→6→5 2→3→6→5 2→4→1→6→5 2→4→3→6→5

III. Phương pháp tham lam

1. Mở đầu

Ý tưởng

Phương pháp tham lam là kĩ thuật thiết kế thường được dùng để giải các bài toán tối ưu. Phương pháp được tiến hành trong nhiều bước. Tại mỗi bước, theo một chọn lựa nào đó (xác định bằng một hàm chọn), sẽ tìm một lời giải tối ưu cho bài toán nhỏ tương ứng. Lời giải của bài toán được bổ sung dần từng bước từ lời giải của các bài toán con.

Các lời giải bằng phương pháp tham lam thường chỉ là chấp nhận được theo điều kiện nào đó, chưa chắc đã tối ưu.

Cho trước một tập A gồm n đối tượng, ta cần phải chọn ra một tập con S của A. Với một tập con S được chọn ra thỏa mãn yêu cầu của bài toán, ta gọi là một nghiệm chấp nhận được. Một hàm mục tiêu gán mỗi nghiệm chấp nhận được với một giá trị. Nghiệm tối ưu là nghiệm chấp nhận được mà tại đó hàm mục tiêu đạt giá trị nhỏ nhất (lớn nhất).

Đặc trưng tham lam của phương pháp thể hiện bởi: trong mỗi bước việc xử lý sẽ tuân theo một sự chọn lựa trước, không kể đến tình trạng không tốt có thể xảy ra.

Mô hình

Chọn S từ tập A.

Tính chất tham lam của thuật toán được định hướng bởi hàm chọn

Khởi động: S = rỗng

Trong khi A khác rỗng

Chọn phần tử tối ưu nhất của A gán vào x : x = chọn(A);

Cập nhật các đối tượng để chọn: A = A-{x};

Nếu S$\cup$ {x} thỏa mãn yêu cầu của bài toán thì cập nhật lời giải: S = S$\cup$ {x};

Thủ tục tham lam

```
Input: A[1..n]
Output: lời giải S
Greedy(A,n)
{
        S = Rỗng;
        While(A ≠ Rỗng)
        {
                A = A-{x};
```

If($S \cup \{x\}$ chấp nhận được)

$S = S \cup \{x\};$

}

Return S;

}

2. Bài toán người du lịch

Phát biểu bài toán

Một người du lịch muốn tham quan n thành phố T_1, .., T_n. Xuất phát từ một thành phố nào đó và đi qua tất cả các thành phố còn lại, mỗi thành phố qua đúng một lần và quay về thành phố xuất phát.

Gọi C_{ij} là chi phí chi từ thành phố T_i đến thành phố T_j. Hãy tìm một hành trình thỏa yêu cầu bài toán với chi phí nhỏ nhất.

Ý tưởng

Đây là bài toán tìm chu trình có trọng số nhỏ nhất trong một đồ thị đơn có hướng có trọng số.

Thuật toán tham lam cho bài toán là chọn thành phố có chi phí nhỏ nhất tính từ thành phố hiện thời đến các thành phố chưa đi qua.

Mô tả thuật toán

Input: $C = C_{ij}$

Output: TOUR //Hành trình tối ưu

COST //Chi phí tương ứng

TOUR = 0;

COST = 0; v = u;

$\forall k = 1 \rightarrow n$

Chọn <v,w> là đoạn nối hai thành phố có chi phí nhỏ nhất tính từ thành phố v đến các thành phố chưa qua.

TOUR = TOUR + <v,w>;

COST = COST + C_{vw}

Hoàn thành chuyến đi

TOUR = TOUR + <v,u>;

COST = COST + C_{vu}

Độ phức tạp thuật toán

Thao tác chọn đỉnh thích hợp trong n đỉnh được tổ chức bằng một vòng lặp để duyệt, nên chi phí cho thuật toán xác định bởi hai vòng lặp lồng nhau, nên độ phức tạp $T(n) \in O(n^2)$.

Cài đặt

```
int GTS(matra a, int n, int Tour[max], int Ddau)
{
    int v, k, w;
    int min;
    int cost;
    int daxet[max];
    for (int k = 1; k <= n; k++)
            daxet[k] = 0;
    cost = 0;
    int i;
    v = Ddau;
    int i = 1;
    Tour[i] = v;
    daxet[v] = 1;
    while (i < n)
    {
            min = VC;
            for (int k = 1; k <= n; k++)
            {
                    if(daxet[k])
                    if (min > a[v][k])
                    {
                            min = a[v][k];
                            w = k;
                    }
                    v = w;
                    i++;
                    Tour[i] = v;
                    daxet[v] = 1;
```

```
                cost = cost + min;
            }
        }
        cost = cost + a[v][Ddau];
        return cost;
    }
```

3. Thuật toán Prim - Tìm cây bao trùm nhỏ nhất

(trình bày trong chương Đồ thị)

4. Bài toán chiếc túi sách

Phát biểu bài toán

Có n vật, mỗi vật i, $i \in [1..n]$ được đặc trưng bởi trọng lượng w_i(kg) và giá trị sử dụng v_i. Có một chiếc túi xách có khả năng mang m kg. Giả sử $w_i, v_i, m \in N^*, \forall i \in [1..n]$.

Hãy chọn vật xếp vào túi sao cho túi sách thu được giá trị sử dụng lớn nhất.

Các trọng lượng của n vật có thể biểu diễn bởi mảng:

$w = (w_1, w_2, .., v_n)$

Và giá trị sử dụng tượng ứng với các vật:

$v = (v_1, v_2, \ldots, v_n)$

Các vật được chọn được lưu trữ vào một mảng ε với quy ước: $\varepsilon[i] = 1$ tức là vật i được chọn.

Bài toán trở thành:

$$\begin{cases} \sum_{i=1}^{n} \varepsilon_i v_i \to \max \\ \sum_{i=1}^{n} \varepsilon_i v_i \le m \\ \varepsilon_i \in \{0,1\}, \forall i = \overline{1,n} \end{cases}$$

Thiết kế thuật toán

Thuật toán tham lam cho bài toán chọn vật có giá trị giảm dần (theo đơn giá).

Input:

$w = (w_1, w_2, .., v_n)$; //Mảng trọng lượng các vật

$v = (v_1, v_2, \ldots, v_n)$; //Mảng giá trị các vật

m: sức chứa của ba lô

Output:

ε [1..n]; //mảng đánh dấu các vật được chọn

V_{max}: giá trị lớn nhất

Mô tả

Knap_Greedy(w,v,Chon,n,m)

{

Khởi động b[i] = i, $\forall i = \overline{1,n}$; //Lưu trữ chỉ số làm cho mảng giảm dần

Khởi động Chon[i] = 0, $\forall i = \overline{1,n}$;//Mảng đánh dấu vật được chọn

Khởi động V_{max}=0;

Tính đơn giá: $d_i = \frac{v_i}{w_i}$, $\forall i = \overline{1,n}$

For(i=1; i<=n && m>0; i++)

{

j = max(d,n,i); // d[j] = Max{d[i],...,d[n]};

b[i] ↔ b[j]

if(m>w[b[i]])

{

V_{max}+=v[b[i]];

Chon[b[i]] = 1;

M-=w[b[i]];

}

d[i] ↔ d[j]

}

Return V_{max}

}

Độ phức tạp thuật toán

Thuật toán chọn max được sử dụng chính là thuật toán chọn trực tiếp, nên độ phức tạp của thuật toán trong các trường hợp là $O(n^2)$.

Bài tập

1. Cài đặt các thuật toán trình bày trong giáo trình
2. Nhân các số lớn

   Kỹ thuật chia để trị nhân 2 số nguyên dương x, y dưới dạng chuỗi:

   Nhan(x, y)

If(l(x) và l(y) <=4) // chiều dài nhỏ hơn 4

Nhân hai số nguyên kiểu long

Else

Giả sử l(x) = l(y) = n;

Tách thành x hai chuỗi con: a(nửa trái), b(nửa phải)

Tách thành y hai chuỗi con: c(nửa trái), d(nửa phải)

Kq = Nhan(a,c)*10^{n} + Nhan(a,d)*$10^{n/2}$ + Nhan(b,c)*$10^{n/2}$ + Nhan(b,d)

3. Sắp tăng dần một dãy x các số bằng thuật toán trộn tự nhiên:

   Trong khi (số đường chạy của x >1)

   - Tách luân phiên đường chạy của x vào hay dãy trung gian x_1,x_2.
   - Trộn từng cặp đường chạy của x1, x2, lưu vào x

4. Giả sử ổ khóa có n công tắc. Mỗi công tắc ở một trong hai trạng thái đóng hoặc mở. Khóa được mở nếu có ít nhất n/2 công tắc ở trạng thái mở. Liệt kê tất cả các cách mở khóa

5. Một người muốn tham quan qua n thành phố T1, …, Tn. Xuất phát từ một thành phố nào đó, người du lịch muốn đi qua tất cả các thành phố còn lại, mỗi thành phố đi qua đúng một lần rồi quay lại thành phố xuất phát.

   Gọi Cij là chi phí đi từ thành phố Ti đến thành phố Tj.

   - Liệt kê tất cả các hành trình mà người đó có thể đi và kèm theo chi phí tương ứng.
   - Chỉ ra hành trình đi từ thành phố từ Ti đến thành phố Tj thỏa yêu cầu bài toán (nếu có) sao cho chi phí thấp nhất.

6. Cho một lưới hình vuông cấp n, mỗi ô được gán với một số tự nhiên. Tại một ô có thể di chuyển đến ô khác theo hướng lên trên, xuống dưới, qua trái, qua phải.

   Tìm đường đi từ ô đầu tiên (1,1) đến ô (m, m) sao cho tổng các ô đi qua là nhỏ nhất. (1<=m <= n)

7. bài toán đổi tiền xu.

   Có một đồng xu giá trị là n. Hãy đổi thành các đồng xu có giá trị 1 xu, 5 xu, 10 xu, 20 xu, 25 xu sao cho tổng số đồng xu là ít nhất.

Tài liệu tham khảo

1. Alfred V. Aho, John E. Hopcroft và Jeffrey D. Ullman

   "Data Structures and Algorithms"

   Addison Wesley Publishing Company, 1987
2. Donald Knuth, "The art of computer programming"

   Vol1: Fundamental algorithms

   Vol3: Sorting and searching

   Addison Wesley Publishing Company, 1973
3. Niklaus Wirth, "Algorithms + Data structures = programs"

   Prentice Hall INC, 1976
4. Nguyễn Xuân Huy, "Thuật toán", Nhà xuất bản thống kê, Hà Nội, 1988.
5. Trương Chí Tín, giáo trình "Cấu trúc dữ liệu và thuật giải 2", Đại học Đà Lạt, 2002.
6. Nguyễn Văn Linh, Trần Cao Đệ, Trương Thị Thanh Tuyền, Lâm Hoài Bảo, Phan Huy Cường, Trần Ngân Bình, giáo trình "Cấu trúc dữ liệu", Đại học Cần Thơ, 2003 của các tác giả.