

CẤU TRÚC DỮ LIỆU VÀ GIẢI THUẬT

Nguyễn Trần Phúc Thịnh / người trình bày

Mail: thinh.nguyentranphuc@stu.edu.vn

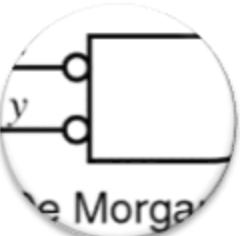
2019

Cấu trúc dữ liệu và giải thuật

CẤU TRÚC DỮ LIỆU VÀ GIẢI THUẬT



CÂY



BẢNG BĂM



ĐỒ THỊ

Cây

Bảng băm

Đồ thị

CÂY

Dạng mệnh đề

- Ôn tập cây (Cây nhị phân tìm kiếm)
- Cây AVL
- Cây tán loe
- Hàng ưu tiên
- B-Cây

Cây

Bảng băm

Đồ thị



Cây

Bảng băm

Đồ thị

Ôn tập cây (Cây nhị phân tìm kiếm)



Cây

Bảng băm

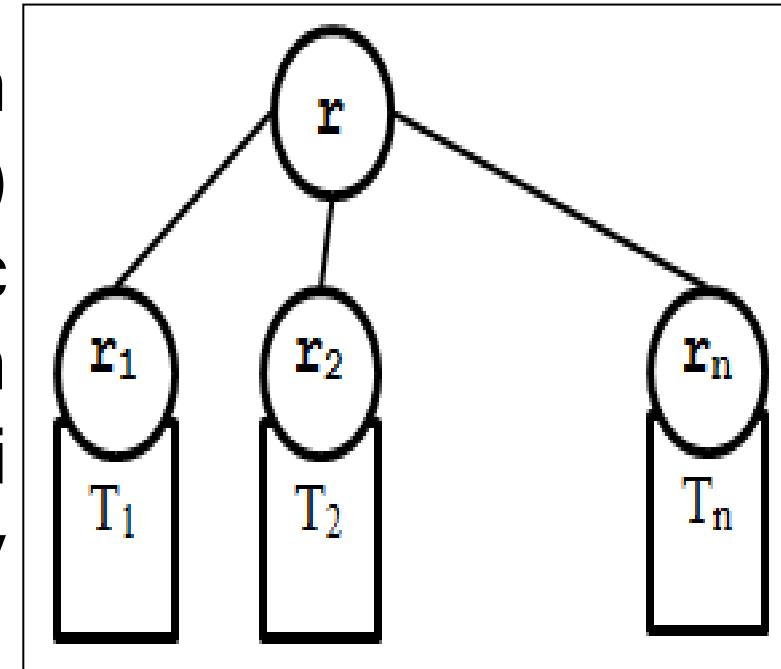
Đồ thị

❑ CÂY

CÂY

1. Khái niệm:

Cây T là tập hợp các phần tử (gọi là các nút của cây) trong đó có một nút đặc biệt r gọi là nút gốc của cây và các nút còn lại được chia ra cho các cây T_1, T_2, \dots, T_n .



- Các cây T_1, T_2, \dots, T_n gọi là các cây con của cây T . Cây T_i có nút gốc r_i ($i=1..n$)

CÂY

Cây

Bảng băm

Đồ thị

- Nút r gọi là nút cha của các nút r_i và nút r_i gọi là nút con của nút r ($i=1..n$).
- Cây không có nút nào gọi là cây rỗng.
- Nút lá là nút không có cây con.
- Nút trung gian (nút nhánh) là nút không phải là nút gốc và nút lá.
- Độ tuổi của nút là số cây con của nút đó.
- Độ tuổi của cây là độ tuổi của nút có độ tuổi lớn nhất.

CÂY

Cây

Bảng băm

Đồ thị

- Bộ (a, b) với a là nút cha của nút b gọi là một cạnh của cây.
- Dãy các nút $a_0, a_1, a_2, \dots, a_n$ gọi là đường đi nếu (a_{i-1}, a_i) là cạnh ($i=1..n$).
- Độ dài đường đi là số cạnh của đường đi.
- Độ cao của nút là độ dài của đường đi dài nhất từ nút đó đến các nút lá.
- Độ cao của cây là độ cao của nút gốc.
- Mức của một nút là độ dài đường đi từ nút gốc đến nút đó.
- Rừng cây là tập hợp nhiều cây.

CÂY

2. Cây nhị phân:

- Khái niệm: Cây nhị phân là cây có bậc là 2.
- Cấu trúc dữ liệu: (*sử dụng con trỏ*)

❑ Ví dụ: Khai báo cây nhị phân có các phần tử là những số nguyên:

```
typedef int TYPEINFO;  
  
struct NODE  
{  
    TYPEINFO data;  
    NODE* left;  
    NODE* right;  
};
```

```
typedef NODE* NODEPTR;  
typedef NODEPTR TREE;
```

CÂY

c. Duyệt cây nhị phân:

- Duyệt theo thứ tự NLR (Node-Left-Right) còn gọi là pre-order:

```
duyetNLR(TREE r)
```

{

- + truy cập nút r
- + duyetNLR(r->left)
- + duyetNLR(r->right)

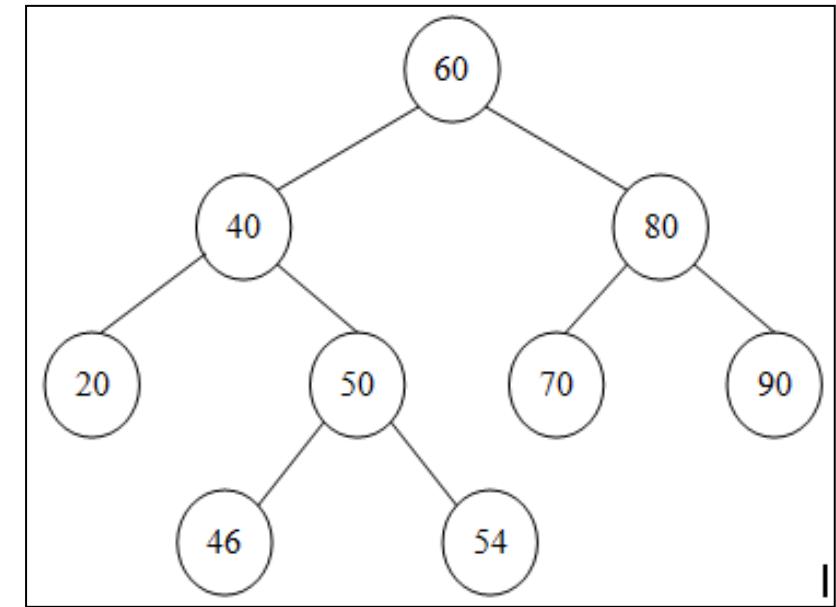
}

- Kết quả duyệt NLR: 60, 40, 20, 50, 46, 54, 80, 70, 90.

Cây

Bảng băm

Đồ thị



CÂY

□ Duyệt theo thứ tự LNR (Left-Node-Right) còn gọi là in-order:

```
duyetLNR (TREE r)
```

```
{
```

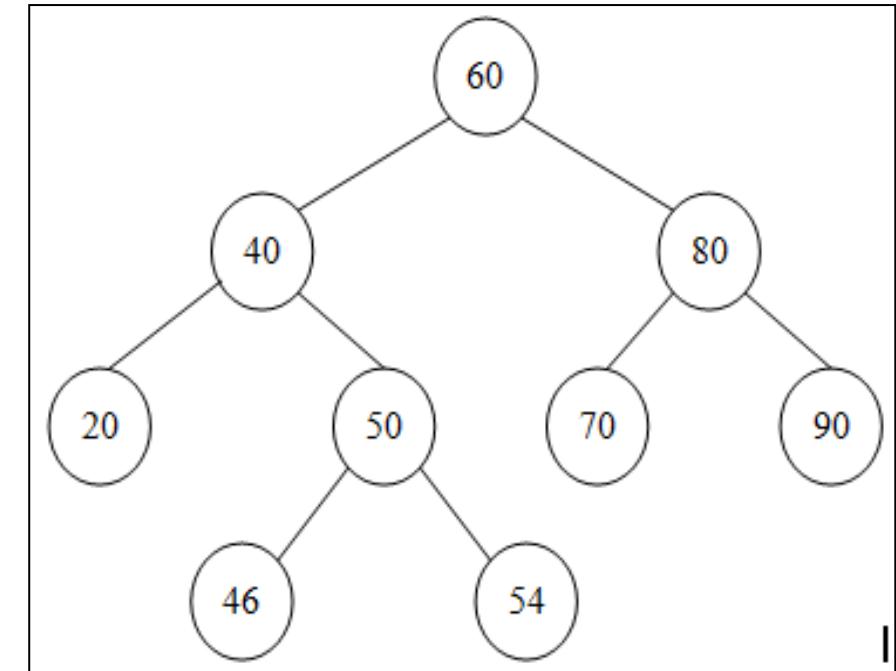
```
    + duyetLNR (r->left)  
    + truy cập nút r  
    + duyetLNR (r->right)
```

```
}
```

Cây

Bảng băm

Đồ thị



- Kết quả duyệt LNR: 20, 40, 46, 50, 54, 60, 70, 80, 90.

CÂY

□ Duyệt theo thứ tự LRN (Left-Right-Node) còn gọi là post-order:

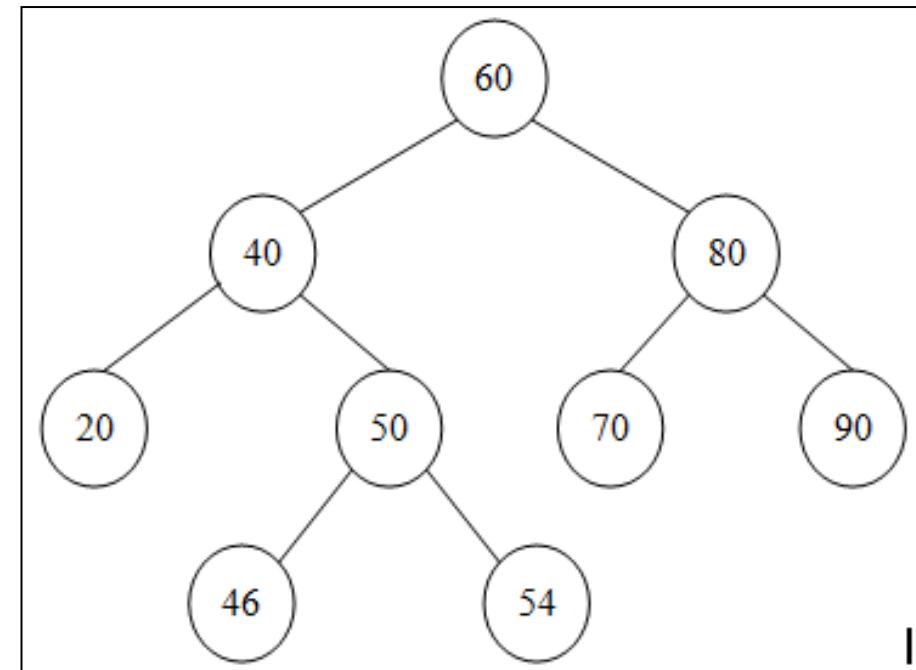
```
duyetLRN (TREE r)
```

```
{  
    + duyetLRN (r->left)  
    + duyetLRN (r->right)  
    + truy cập nút r  
}
```

Cây

Bảng băm

Đồ thị



- Kết quả duyệt LRN: 20, 46, 54, 50, 40, 70, 90, 80, 60.

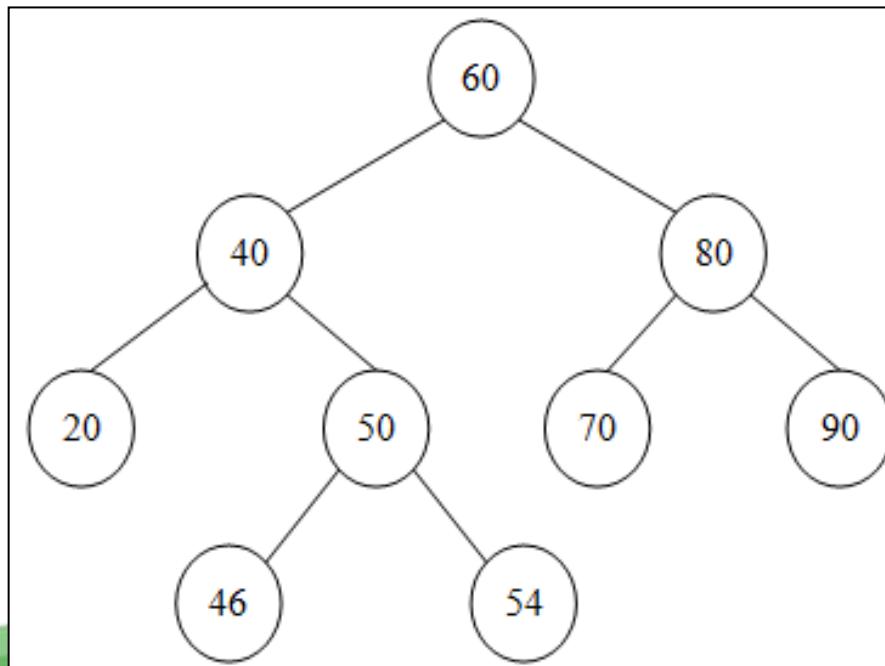
❑ CÂY NHỊ PHÂN TÌM KIẾM

CÂY NHỊ PHÂN TÌM KIẾM

1. Khái niệm:

Cây tìm kiếm nhị phân là cây nhị phân mà với mọi nút của cây thì khoá của nút gốc lớn hơn khoá của mọi nút trong cây con trái và nhỏ hơn khoá mọi nút trong cây con phải.

Ví dụ:



Cây

Bảng băm

Đồ thị

CÂY NHỊ PHÂN TÌM KIẾM

2. Cấu trúc dữ liệu: (giống như cây nhị phân)

```
typedef int TYPEINFO;  
  
struct NODE  
{  
    TYPEINFO data;  
    NODE* left;  
    NODE* right;  
};
```

```
typedef NODE* NODEPTR;  
typedef NODEPTR BST;
```

Nhận xét:

- Nút trái cùng là nhỏ nhất và nút phải cùng là lớn nhất.
- Duyệt cây LNR sẽ được danh sách có thứ tự.

Cây

Bảng băm

Đồ thị

CÂY NHỊ PHÂN TÌM KIẾM

3. Các phép toán:

a. Tìm dữ liệu a trong cây tìm kiếm nhị phân r:

Giải thuật:

If (cây r rỗng)

kết luận không tìm thấy

Else If (khóa của a bằng khóa của nút gốc của r)

kết luận tìm thấy

Else If (khoa của a nhỏ hơn khóa của nút gốc trong cây r)

Tìm a trong cây con trái của cây r

Else Tìm a trong cây con phải của cây r

Cây

Bảng băm

Đồ thị

CÂY NHỊ PHÂN TÌM KIẾM

Cây

Bảng băm

Đồ thị

Ví dụ: Viết hàm tìm nút có dữ liệu a trong cây tìm kiếm nhị phân r. Nếu tìm thấy hàm trả về nút tìm được, ngược lại hàm trả về giá trị NULL.

NODEPTR tim(BST r, TYPEINFO a)

CÂY NHỊ PHÂN TÌM KIẾM

Cây

Bảng băm

Đồ thị

b. Chèn dữ liệu a vào cây tìm kiếm nhị phân:

Giải thuật:

If (cây r rỗng)

- + Tạo nút p chứa dữ liệu a
- + $r = p$

Else If (khoá của a nhỏ hơn khoá của nút gốc trong cây r)

Chèn a vào cây con trái của cây r

Else If (khoá của a lớn hơn khoá của nút gốc trong cây r)

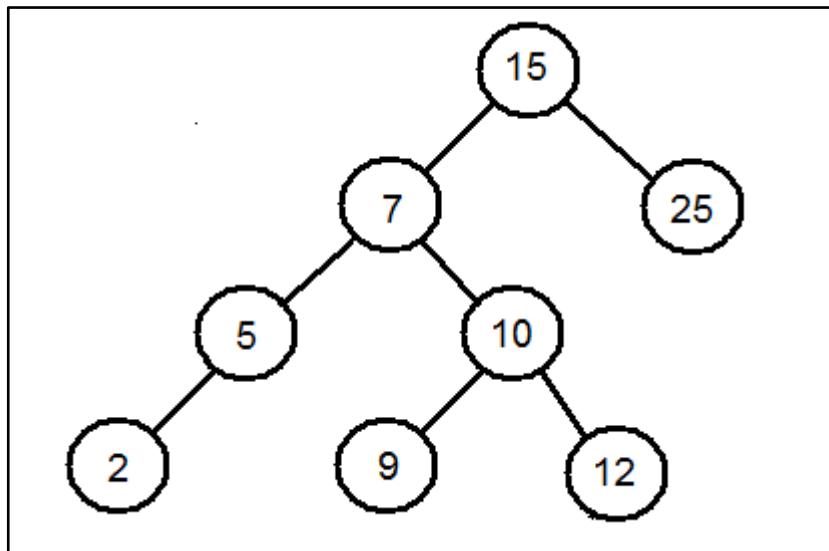
Chèn a vào cây con phải của cây r

CÂY NHỊ PHÂN TÌM KIẾM

Ví dụ: Viết hàm chèn nút có dữ liệu a vào cây tìm kiếm nhị phân r.

```
void chen(BST& r, TYPEINFO a)
```

c. Loại bỏ nút có dữ liệu a trong cây tìm kiếm nhị phân r:



Cây

Bảng băm

Đồ thị

CÂY NHỊ PHÂN TÌM KIẾM

c. Loại bỏ nút có dữ liệu a trong cây tìm kiếm nhị phân r:

Giải thuật:

Bước 1: Tìm nút p có dữ liệu a trong cây r

Bước 2: Nếu tìm thấy thì loại bỏ nút p.

Giải thuật (chi tiết của Bước 2):

Bước 2.1: Nếu p là node lá thì xóa p, ngược lại
Tìm q là nút tận cùng bên phải của cây con trái
của cây p (hay là nút tận cùng bên trái của cây
con phải của cây p).

Bước 2.2: Sao chép dữ liệu của nút q sang nút p.

Bước 2.3: Loại bỏ nút q. (đây là bước đệ quy lại
bước 2.1, lúc này q chính là p mới cần loại bỏ)

Cây

Bảng băm

Đồ thị

CÂY NHỊ PHÂN TÌM KIẾM

Cây

Bảng băm

Đồ thị

Ví dụ: Viết hàm loại bỏ nút có dữ liệu a trong cây tìm kiếm nhị phân r.

```
void xoa(BST& r, TYPEINFO a)
```

Bài tập:

Phần 1:

- 1) Hãy chèn lần lượt các phần tử sau vào cây tìm kiếm nhị phân T rỗng: 23, 45, 9, 12, 20, 17, 10, 38, 30.
- 2) Loại bỏ phần tử 23 của cây T.
- 3) Loại bỏ phần tử 9 của cây T.
- 4) Ghi kết quả duyệt cây trong câu 1/ theo NLR, LNR, và LRN.

III. BÀI TẬP

Bài tập

Phần 2: cho r là cây nhị phân. Viết hàm:

- 1) Đếm số nút có trong cây r.
- 2) Đếm số nút lá có trong cây r.
- 3) Độ cao của cây r.
- 4) Mức của nút p trong cây r.

Cây

Bảng băm

Đồ thị



Cây

Bảng băm

Đồ thị

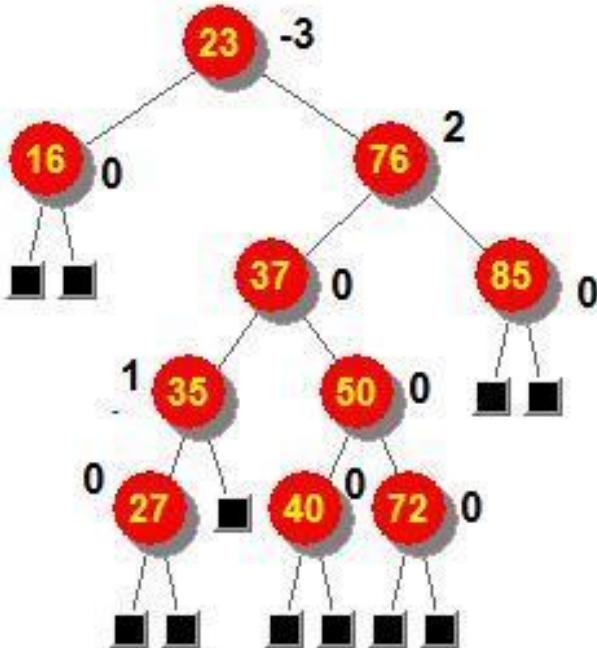
Cây AVL (Cây cân bằng)

CÂY CÂN BẰNG

Cây

Bảng băm

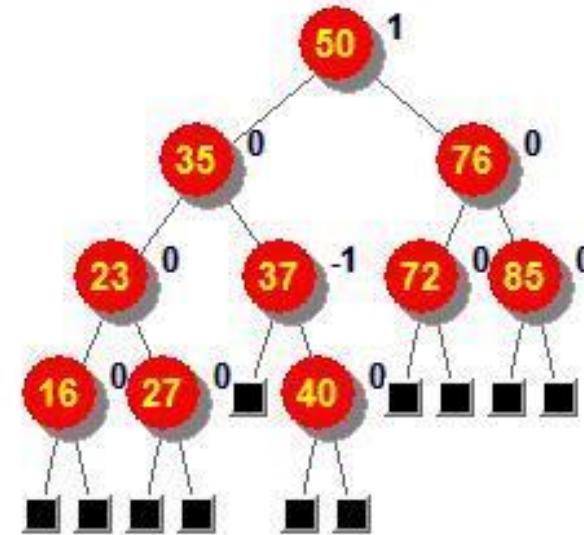
Đồ thị



1. Cây tìm kiếm nhị phân
không là cây AVL

Hai cây được tạo từ cùng dãy khóa

23;76;37;85;50;40;72;35;16;27;



2. Cây AVL

CÂY CÂN BẰNG

1. Khái niệm:

Một cây AVL là một cây tìm kiếm nhị phân tự cân bằng, và là cấu trúc dữ liệu đầu tiên có khả năng này. Trong một cây AVL (G.M. Adelson-Velsky và E.M. Landis), tại mỗi nút chiều cao của hai cây con sai khác nhau không quá một. Hiệu quả là các phép chèn (insertion), và xóa (deletion) luôn chỉ tốn thời gian $O(\log n)$ trong cả trường hợp trung bình và trường hợp xấu nhất. Phép bổ sung và loại bỏ có thể cần đến việc tái cân bằng bằng một hoặc nhiều phép xoay.

Cây

Bảng băm

Đồ thị

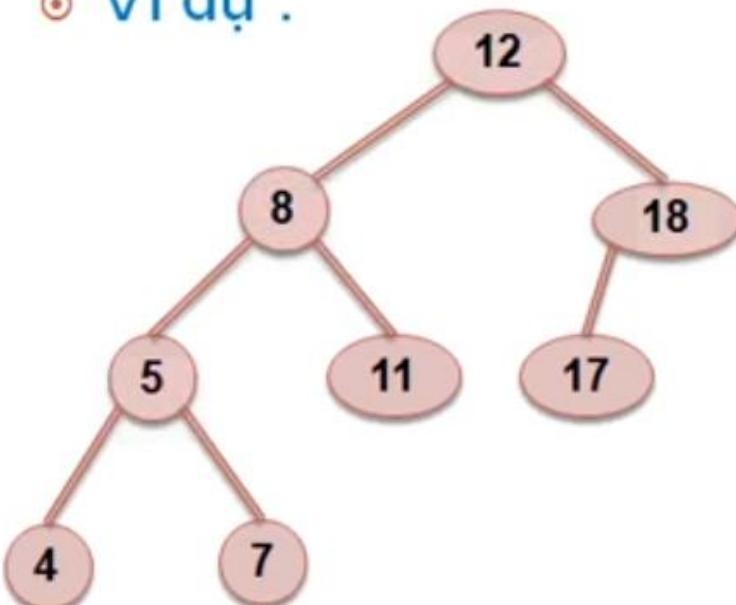
CÂY CÂN BẰNG

Cây

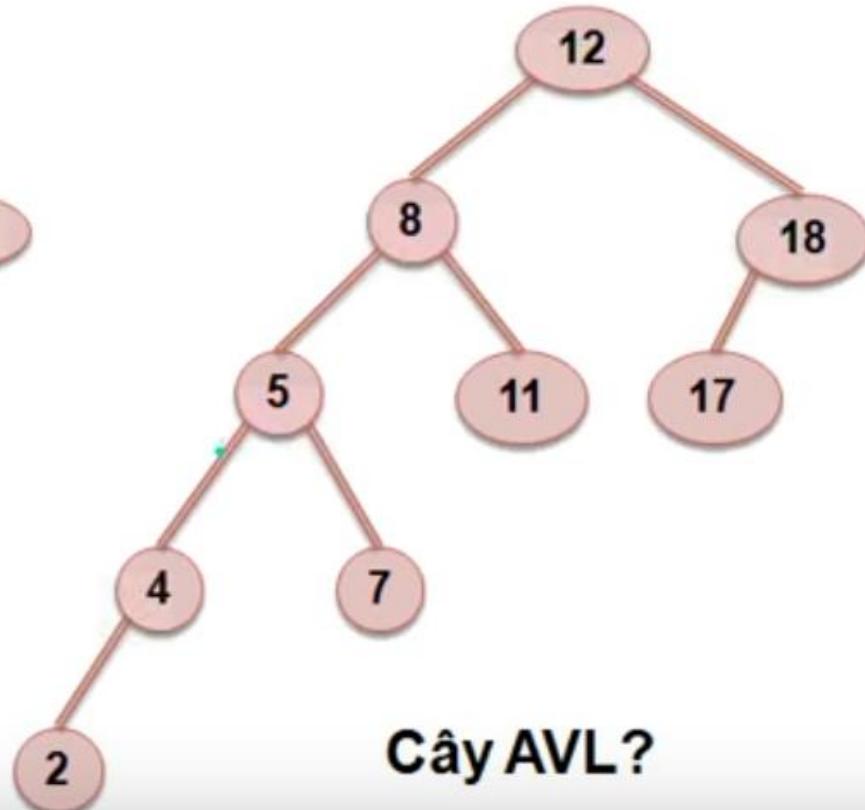
Bảng băm

Đồ thị

• Ví dụ :



Cây AVL?



Cây AVL?

CÂY CÂN BẰNG

- Để biểu diễn cây cân bằng, ta thêm một thuộc tính mới (height) vào mỗi node của cây thể hiện thể hiện chiều cao của node. Quy ước rằng node null có chiều cao bằng không.
- Cấu trúc:

```
typedef struct NODEAVL{  
    TYPEINFO info;  
    struct NODEAVL *left;  
    struct NODEAVL *right;  
    int height;  
};  
typedef NODEAVL* AVLTREE;
```

Cây

Bảng băm

Đồ thị

CÂY CÂN BẰNG

Cây

Bảng băm

Đồ thị

2. Các trường hợp cây AVL mất cân bằng:

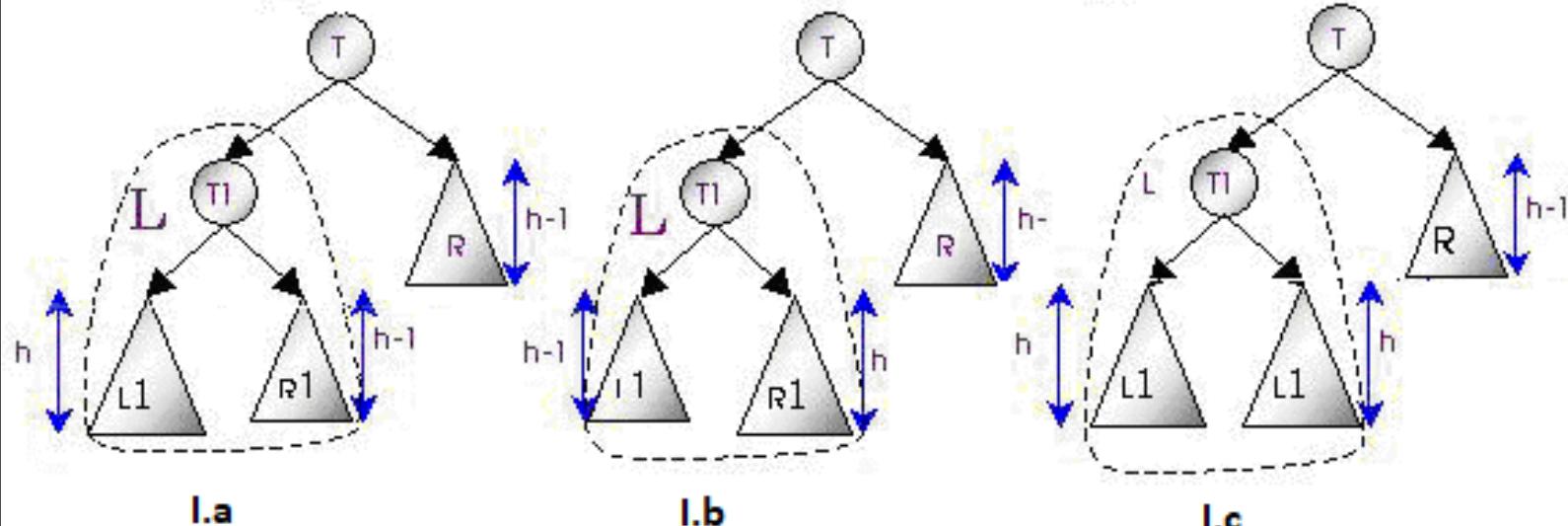
Đôi lúc sau khi thêm hoặc xóa khóa s khỏi cây T thì cây T mất cân bằng. Có 6 khả năng sau:

- I. Cây T lệch về bên trái: (cây con trái cao hơn cây con phải)
 - a. Cây con trái của T lệnh trái.
 - b. Cây con trái của T lệnh phải.
 - c. Cây con trái của T cân bằng.
- II. Cây T lệch về bên phải: (cây con phải cao hơn cây con trái)
 - a. Cây con phải của T lệnh phải.
 - b. Cây con phải của T lệnh trái.
 - c. Cây con phải của T cân bằng.

❑ CÂY AVL MẤT CÂN BẰNG LỆCH TRÁI

CÂY CÂN BẰNG

I. Cây T lệnh trái:



Cây

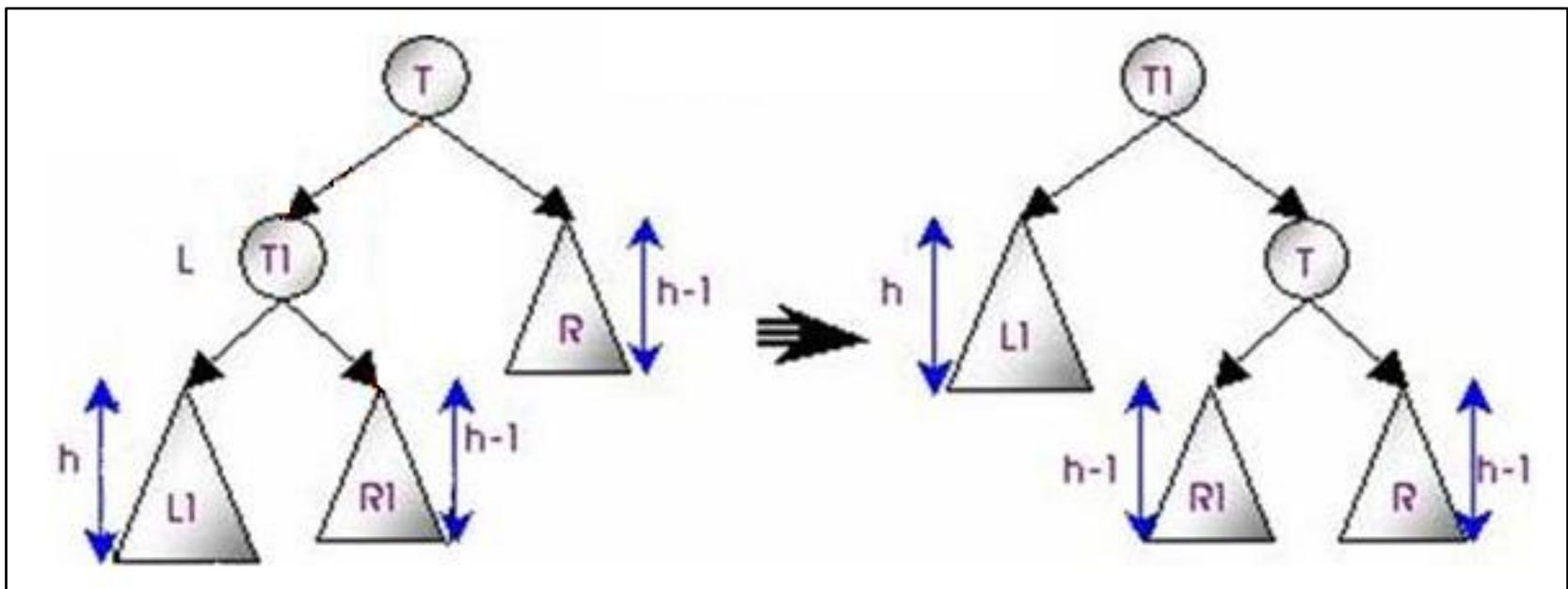
Bảng băm

Đồ thị

CÂY CÂN BẰNG

I.a. Mất cân bằng trái-trái (LL)

Một cây T được gọi là mất cân bằng trái-trái khi T lệch trái và T_1 là cây con trái của T thì T_1 cũng lệch trái.

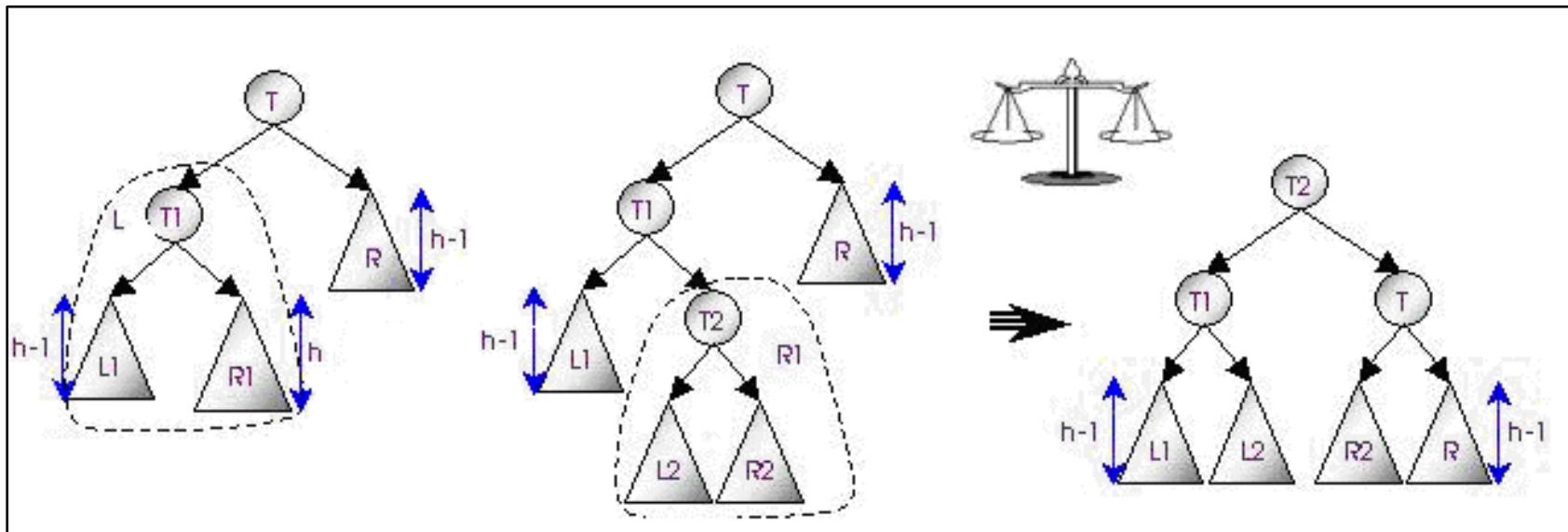


Thực hiện phép quay phải tại T (quay đơn left-left)

CÂY CÂN BẰNG

I.b. Mất cân bằng trái-phải (LR)

Một Cây T được gọi là mất cân bằng trái phải khi T lêch trái và T_1 là cây con trái của T thì T_1 lêch phải.



Thực hiện phép quay trái tại T_1 để đưa về TH LL sau đó thực hiện phép quay phải tại T (quay kép left-right)

Cây

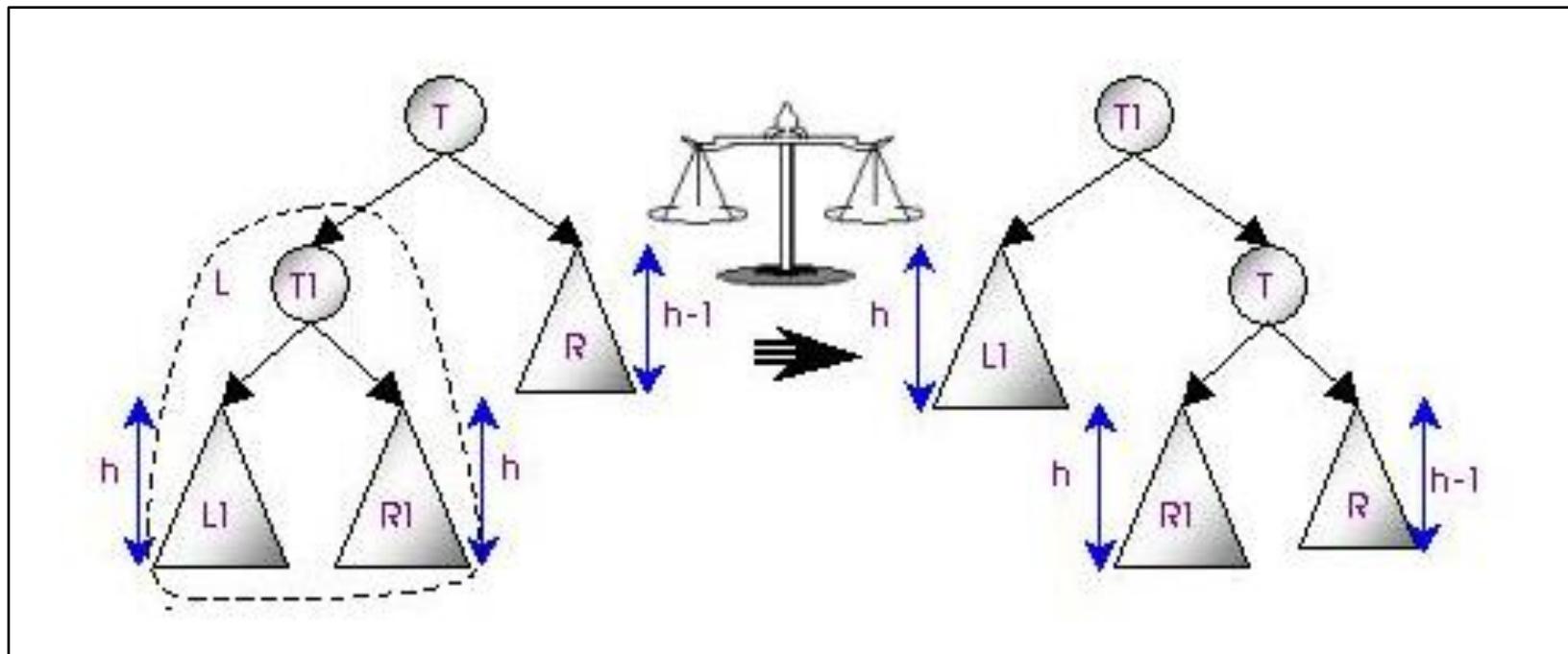
Bảng băm

Đồ thị

CÂY CÂN BẰNG

I.c. Mất cân bằng trái-cân bằng (LE)

Một đỉnh u được gọi là mất cân bằng trái - cân bằng khi u lệch trái và v là cây con trái của u thì v cân bằng.

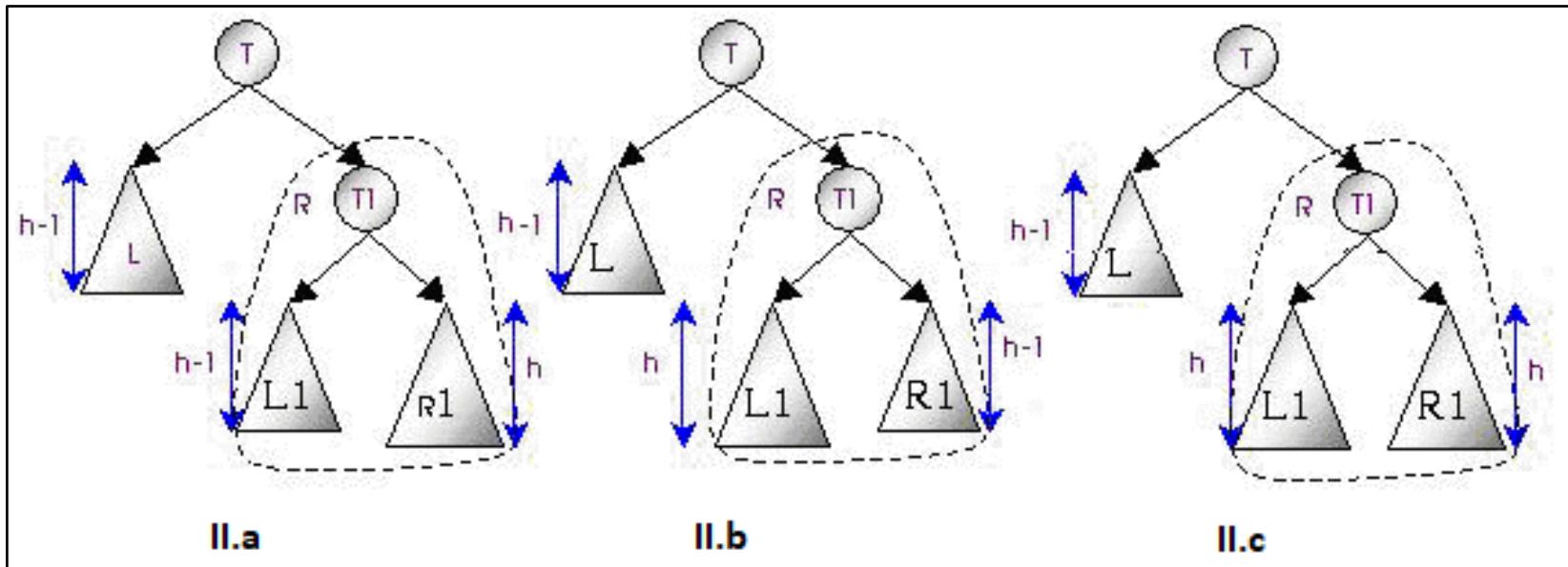


Thực hiện phép quay phải tại T(quay đơn left left)

❑ CÂY AVL MẤT CÂN BẰNG LỆCH PHẢI

CÂY CÂN BẰNG

II. Cây T lệnh phải:



Các trường hợp này đối xứng với các trường hợp khi cây T lệch trái nên việc cân bằng lại cây cũng đối xứng tương tự.

❑ CÁC PHÉP QUAY CÂN BẰNG CÂY

CÂY CÂN BẰNG

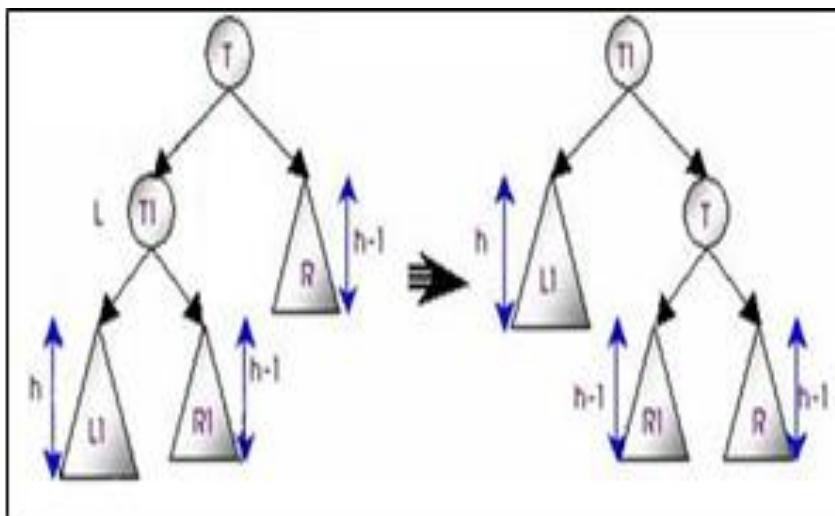
Cây

Bảng băm

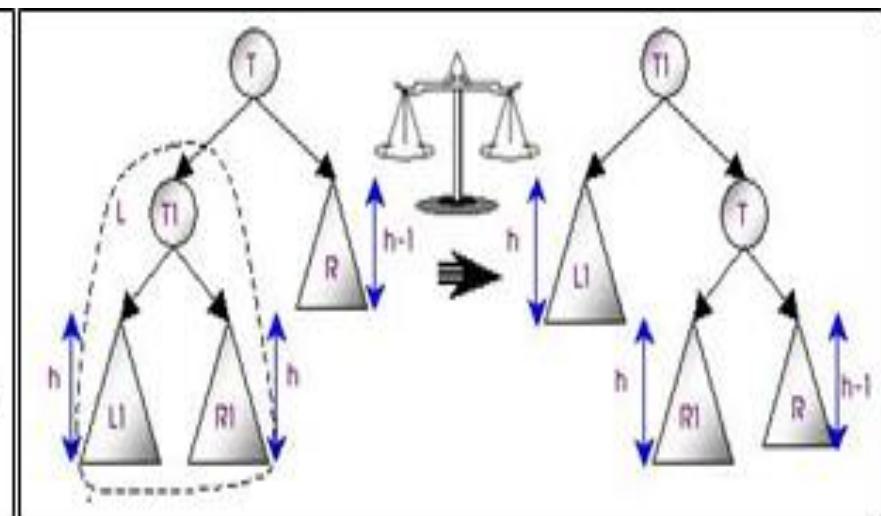
Đồ thị

➤ Phép quay phải (quay đơn left-left)

Gọi T là node bị mất cân bằng (T lệch trái) và T_1 là cây con trái của T



Trường hợp 1: T_1 lệch trái



Trường hợp 2: T_1 cân bằng

CÂY CÂN BẰNG

➤ Phép quay phải (quay đơn left-left)

Giải thuật:

Bước 1: Xoay

- T trỏ trái = T1 trỏ phải
- T1 trỏ phải = T

Bước 2: Cập nhật chiều cao

- Cập nhật lại chiều cao cho node T
- Cập nhật lại chiều cao cho node T1

Bước 3 : Gán lại node gốc

- Node T mới chính là node T1

Cây

Bảng băm

Đồ thị

CÂY CÂN BẰNG

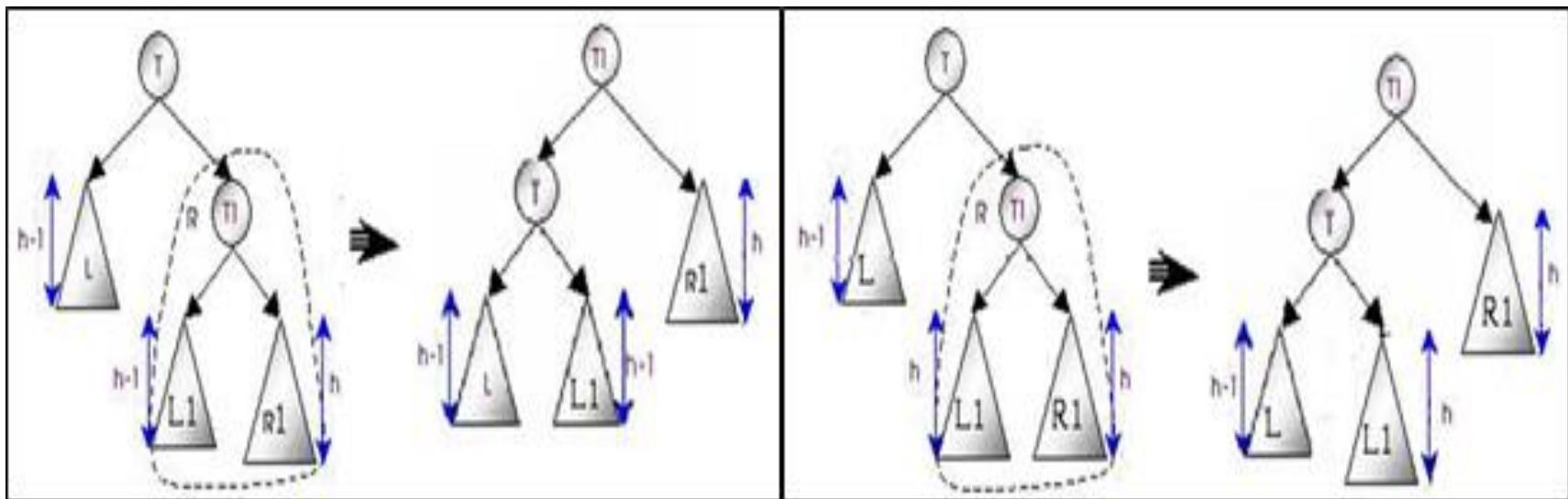
Cây

Bảng băm

Đồ thị

➤ Phép quay trái (quay đơn right-right)

Gọi T là node bị mất cân bằng (T lệch phải) và T_1 là cây con phải của T



Trường hợp 1: T_1 lệch phải

Trường hợp 2: T_1 cân bằng

CÂY CÂN BẰNG

➤ Phép quay trái (quay đơn right-right)

Giải thuật:

Bước 1: Xoay

- T trỏ phải = T1 trỏ trái
- T1 trỏ trái = T

Bước 2: Cập nhật chiều cao

- Cập nhật lại chiều cao cho node T
- Cập nhật lại chiều cao cho node T1

Bước 3 : Gán lại node gốc

- Node T mới chính là node T1

Cây

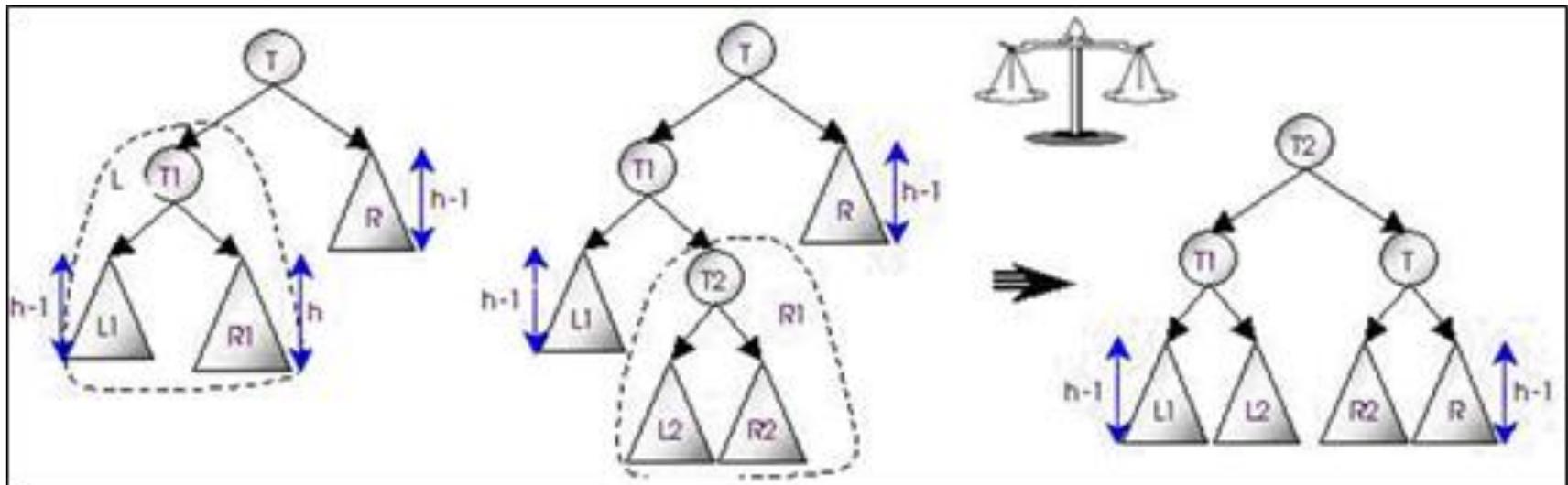
Bảng băm

Đồ thị

CÂY CÂN BẰNG

➤ Phép quay kép left-right

Gọi T là node bị mất cân bằng (T lệch trái), T_1 là cây con trái của T , T_1 lệch phải. Gọi T_2 là cây con phải của T_1 .



CÂY CÂN BẰNG

Bước 1: Xoay

- T trỏ trái = T_2 trỏ phải
- T_2 trỏ phải = T
- T_1 trỏ phải = T_2 trỏ trái
- T_2 trỎ trái = T_1

Bước 2: Cập nhật chiều cao

- Cập nhật lại chiều cao cho node T
- Cập nhật lại chiều cao cho node T_1
- Cập nhật lại chiều cao cho node T_2

Bước 3 : Gán lại node gốc

- Node T mới chính là node T_2

Cây

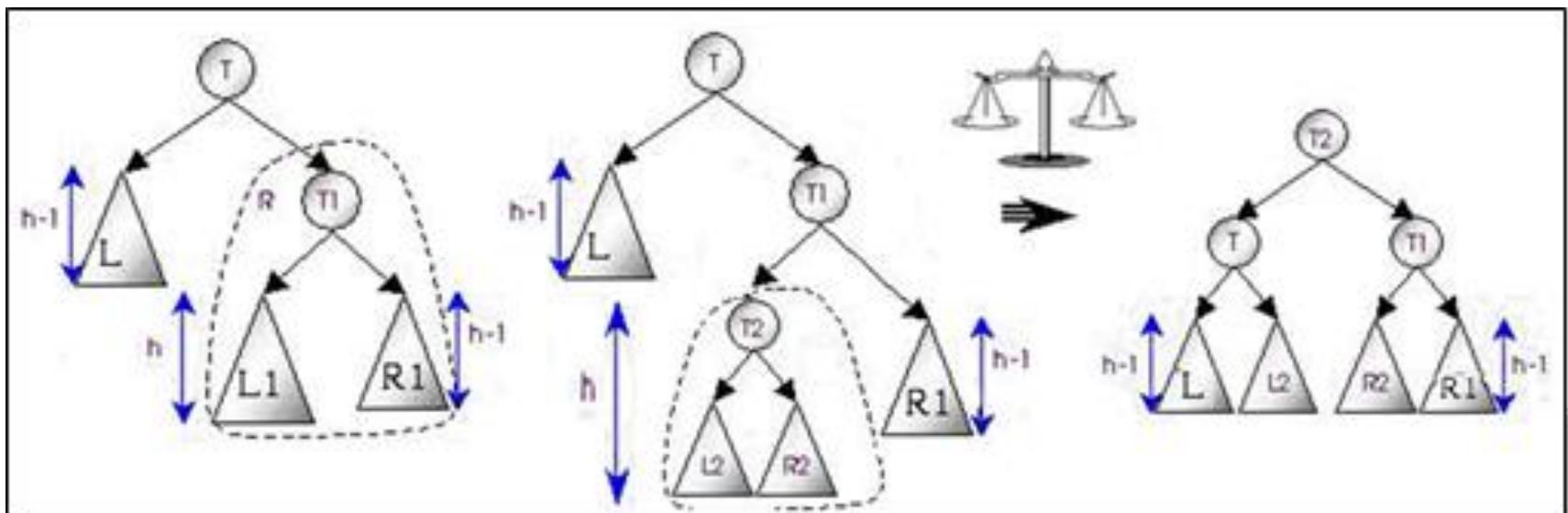
Bảng băm

Đồ thị

CÂY CÂN BẰNG

➤ Phép quay kép right-left

Gọi T là node bị mất cân bằng (T lệch phải), T_1 là cây con phải của T , T_1 lệch trái. Gọi T_2 là cây con trái của T_1 .



CÂY CÂN BẰNG

Bước 1: Xoay

- T trỏ phải = T2 trỏ trái
- T2 trỎ trái = T
- T1 trỎ trái = T2 trỎ phải
- T2 trỎ phải = T1

Bước 2: Cập nhật chiều cao

- Cập nhật lại chiều cao cho node T
- Cập nhật lại chiều cao cho node T1
- Cập nhật lại chiều cao cho node T2

Bước 3 : Gán lại node gốc

- Node T mới chính là node T2

Cây

Bảng băm

Đồ thị

CÂY CÂN BẰNG

Cây

Bảng băm

Đồ thị

3. Các phép toán trên cây AVL:

- Tìm một khóa trong cây AVL
- Thêm một dữ liệu vào Cây AVL
- Xóa một dữ liệu khỏi Cây AVL

☐ Tìm một khóa trong cây AVL

- Việc tìm một dữ liệu có trong cây AVL hay không được thực hiện tương tự như tìm kiếm trong cây nhị phân thông thường.

❑ THÊM MỘT DỮ LIỆU VÀO CÂY AVL

CÂY CÂN BẰNG

Cây

Bảng băm

Đồ thị

Việc chèn một dữ liệu X vào cây AVL T thực hiện tương tự như cây nhị phân tìm kiếm. Tuy nhiên khi thêm một node mới vào cây có thể dẫn đến việc mất cân bằng tại một node nào đó trong cây. Khi đó, tại node mất cân bằng gần node vừa thêm vào, tùy theo trường hợp mất cân bằng mà ta cần thực hiện các phép quay đã giới thiệu trong phần trên để cân bằng lại cây.

CÂY CÂN BẰNG

Để thuận tiện trong việc kiểm tra độ cân bằng của 1 node T ta xây dựng thêm 1 hàm GetBalance(AVLTREE T), hàm này trả kết quả hiệu chiều cao cây con trái của T và chiều cao cây con phải của T.

Cây

Bảng băm

Đồ thị

GetBalance (T)	Ý nghĩa	Ghi chú
2	T Mất cân bằng trái	Cần thực hiện các phép quay để tái cân bằng
1	T Lệch trái	
0	T cân bằng	Không cần xử lý
-1	T lệch phải	
-2	T mất cân bằng phải	Cần thực hiện các phép quay để tái cân bằng

CÂY CÂN BẰNG

Chúng ta xây dựng một hàm InsertNode để chèn một dữ liệu X vào cây AVL T. Hàm trả về:

NULL: Đã có dữ liệu trong cây (thêm thất bại)

AVLTREE: Node chứa dữ liệu X (Thêm thành công)

Cây

Bảng băm

Đồ thị

CÂY CÂN BẰNG

Nguyên mẫu hàm

InsertNode(AVLTREE &T, TYPEINFO X)

- Đầu vào:
 - Một cây cân bằng T
 - Một khóa X cần chèn
- Đầu ra:
 - NULL: Đã có dữ liệu trong cây (thêm thất bại)
 - AVLTREE: Node chứa dữ liệu X (Thêm thành công)

Cây

Bảng băm

Đồ thị

Giải thuật

B1: Chèn X vào cây T

Nếu cây rỗng

T là node mới chứa thông tin X.

Trả về T

Nếu T chứa khóa là X

Trả về giá trị NULL

Nếu T chứa khóa lớn hơn X

result = chèn X vào cây con trái của T

Ngược lại

result = chèn X vào cây con phải của T

B2: Cập nhật lại chiều cao của cây T

B3: Kiểm tra và tái cân bằng

Sử dụng hàm GetBalance để kiểm tra sự cân bằng của cây T

Tùy vào tình trạng mất cân bằng mà thực hiện các phép quay tương ứng

B4: Trả về kết quả

Return result;

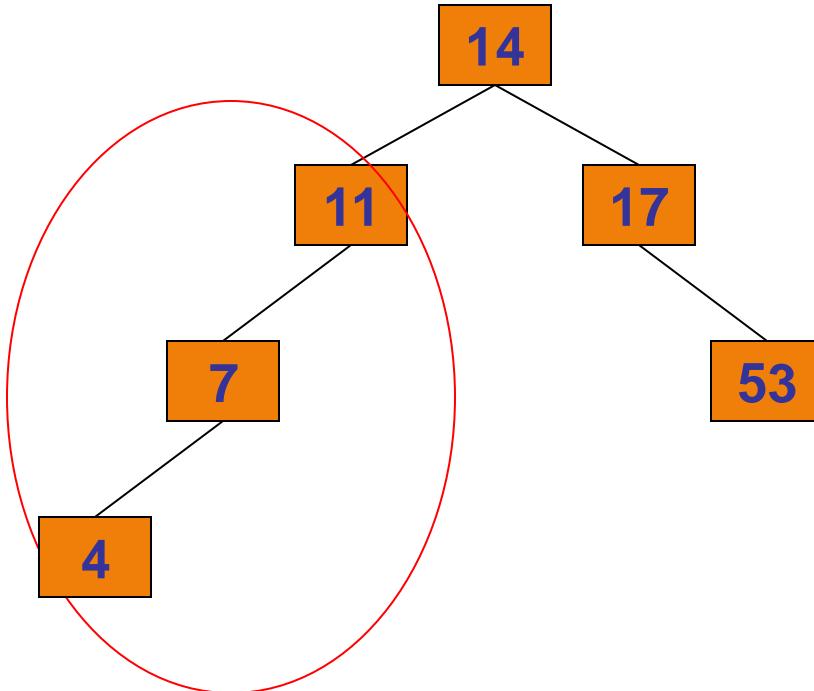
Cây

Bảng băm

Đồ thị

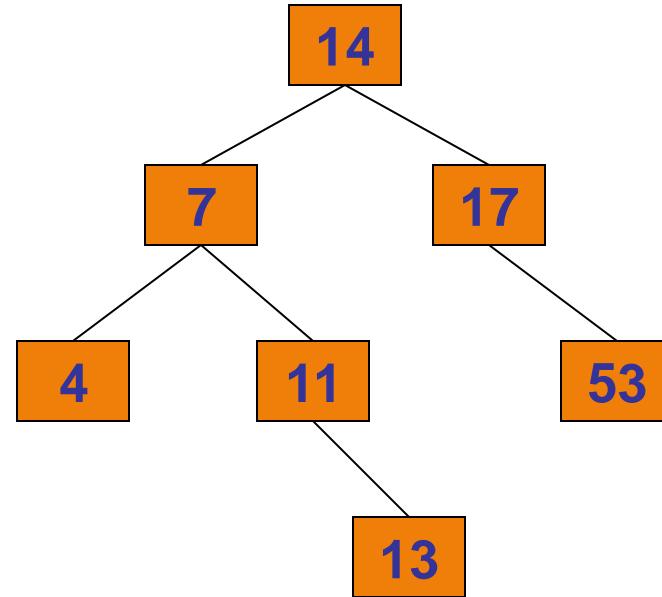
Ví dụ về cây AVL:

chèn 14, 17, 11, 7, 53, 4, 13 vào 1 cây AVL rỗng



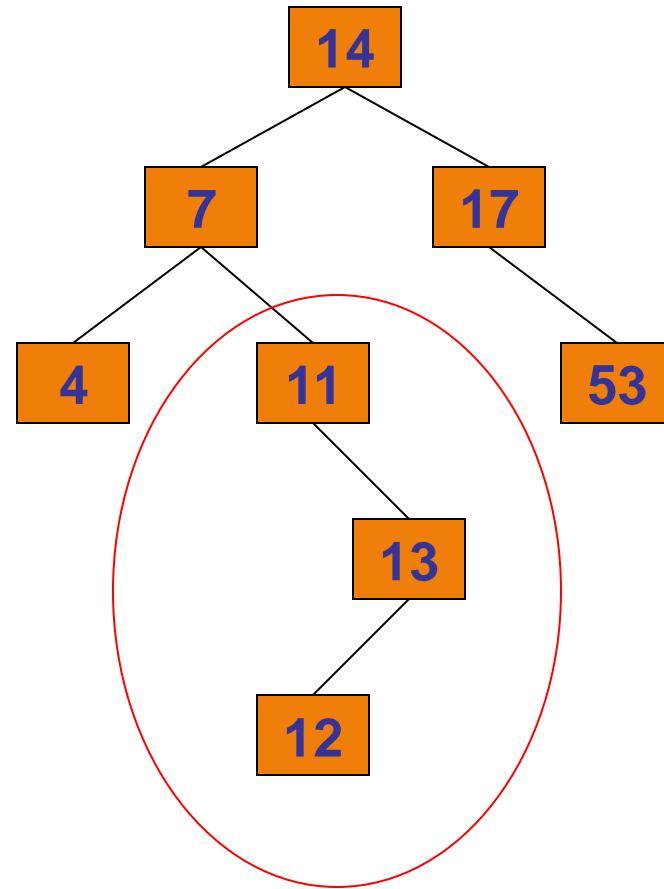
Ví dụ về cây AVL:

chèn 14, 17, 11, 7, 53, 4, 13 vào 1 cây AVL rỗng



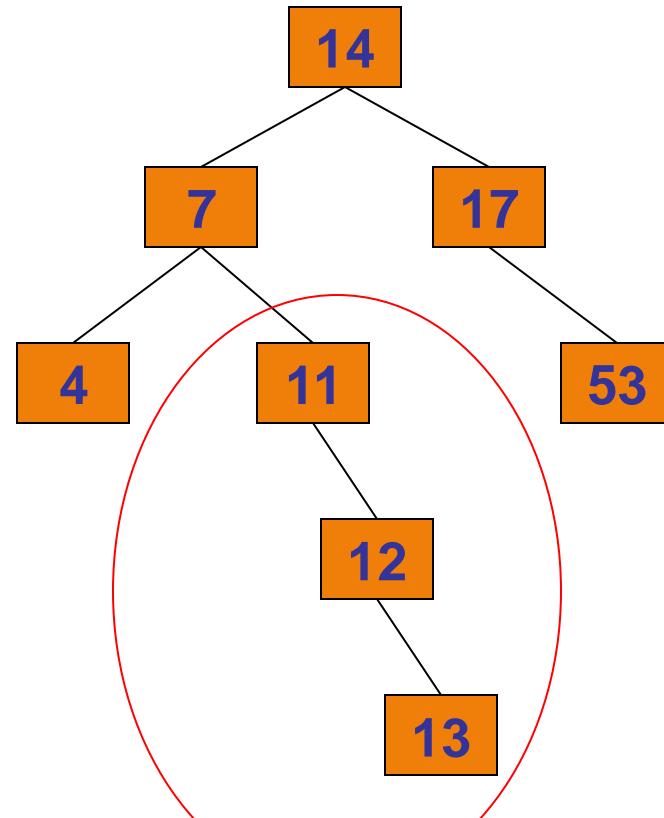
Ví dụ cây AVL:

- Chèn tiếp 12



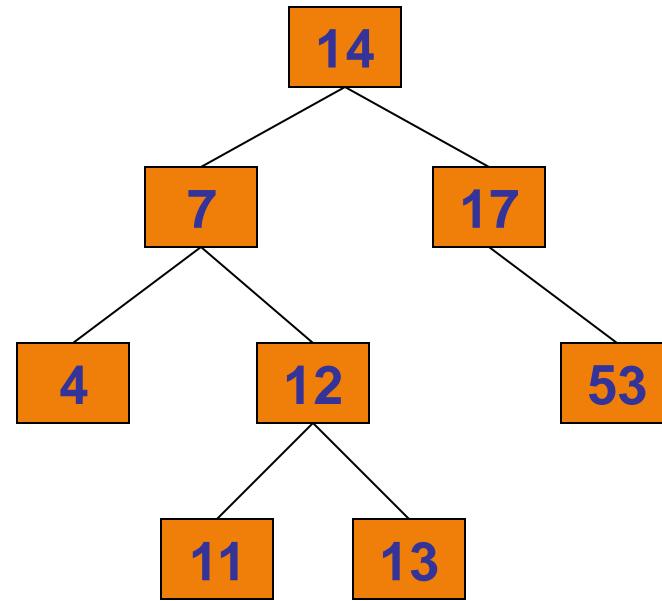
Ví dụ cây AVL:

- Chèn tiếp 12



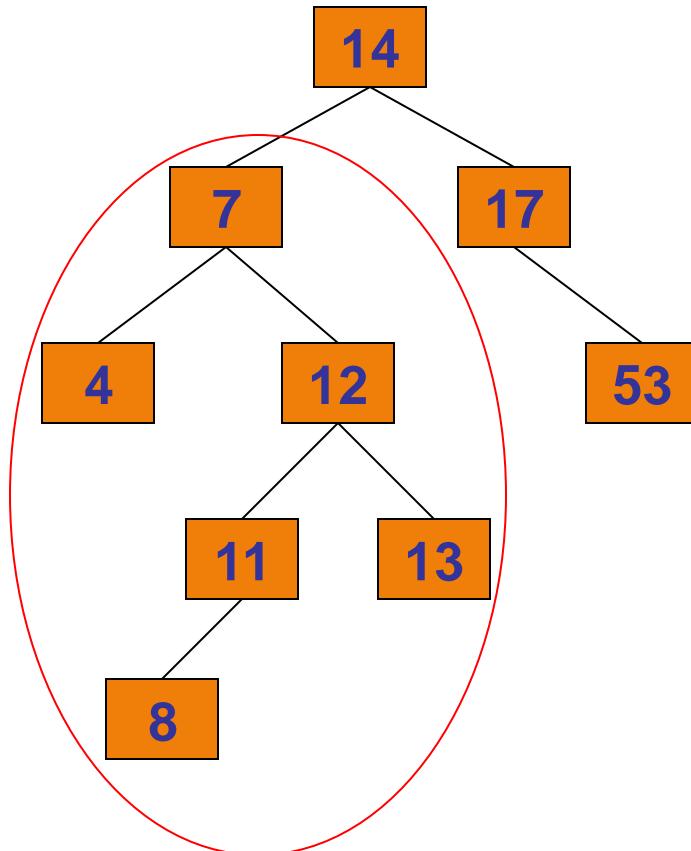
Ví dụ cây AVL:

- cây đã cân bằng sau khi xoay gi?



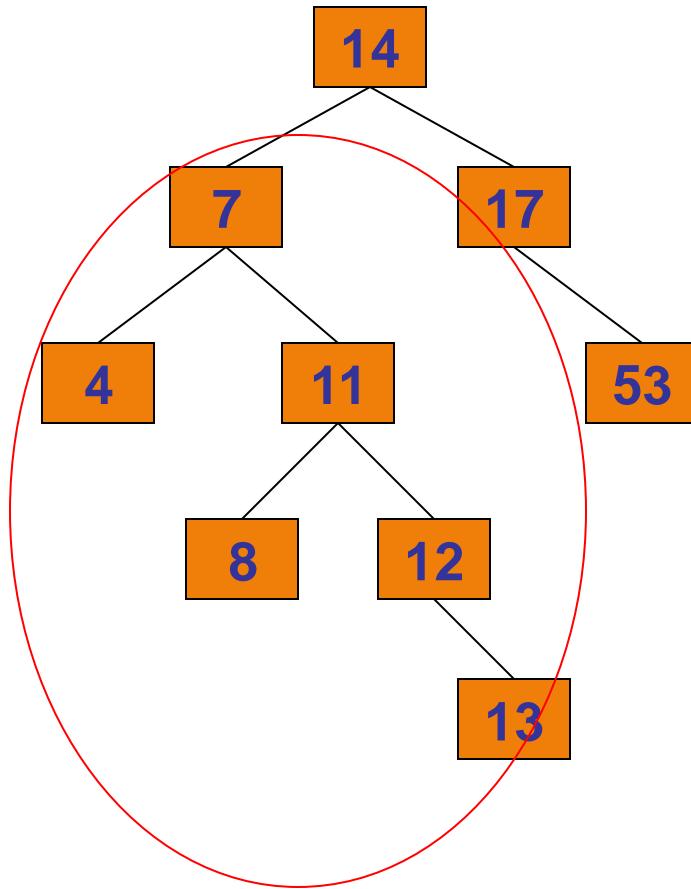
Ví dụ cây AVL:

- Chèn tiếp 8



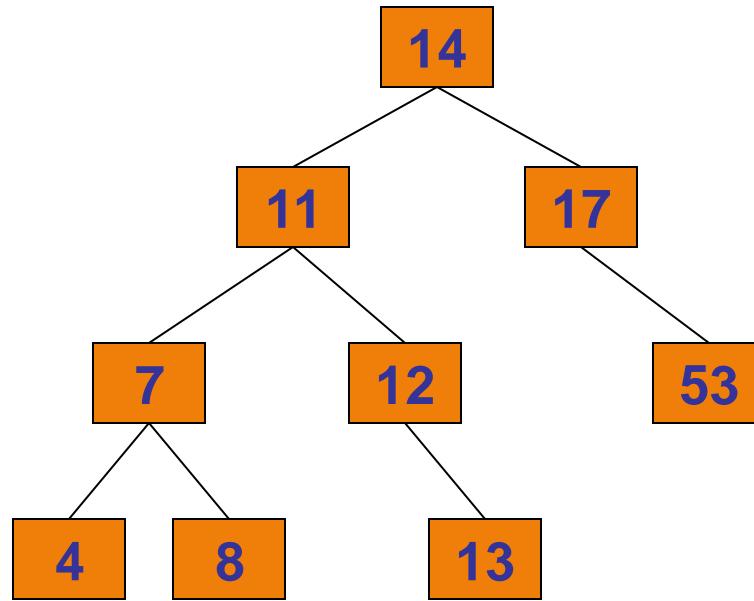
Ví dụ cây AVL:

- Chèn tiếp 8



Ví dụ cây AVL:

- cây đã cân bằng



XÓA MỘT DỮ LIỆU KHỎI CÂY AVL

CÂY CÂN BẰNG

Khi xóa bỏ một dữ liệu X trong cây AVL T cũng có khả năng làm một số node trên cây bị mất cân bằng. Khi đó tại node mất cân bằng gần node vừa xóa ta thực hiện các phép quay tương ứng để cân bằng lại cây.

Chúng ta xây dựng một hàm DeleteNode với các giá trị trả về:

- 0: Không có dữ liệu cần xóa trong cây
- 1: Xóa thành công.

Cây

Bảng băm

Đồ thị

CÂY CÂN BẰNG

Nguyên mẫu hàm

DeleteNode(AVLTREE &T, TYPEINFO X)

- Đầu vào:
 - Một cây cân bằng T
 - Một khóa X cần xóa
- Đầu ra:
 - 0: Không có dữ liệu cần xóa trong cây
 - 1: Xóa thành công.

Cây

Bảng băm

Đồ thị

Giải thuật

B1: Xóa X khỏi cây T

result ban đầu bằng 1

Nếu cây rỗng

Trả về 0

Nếu T chứa khóa lớn hơn X

result = xóa X trong cây con trái của T

Ngược lại Nếu T chứa khóa nhỏ hơn X

result = xóa X trong cây con phải của T

Ngược lại lúc này T chứa khóa X

Xóa node T (thực hiện như xóa node trong cây BST)

B2: Nếu T khác rỗng

B2.1 : Cập nhật lại chiều cao của cây T

B2.2 : Kiểm tra và tái cân bằng

Sử dụng hàm GetBalance để kiểm tra sự cân bằng của cây T. Tùy vào tình trạng mất cân bằng mà thực hiện các phép quay tương ứng

B3: Trả về kết quả

Return result;

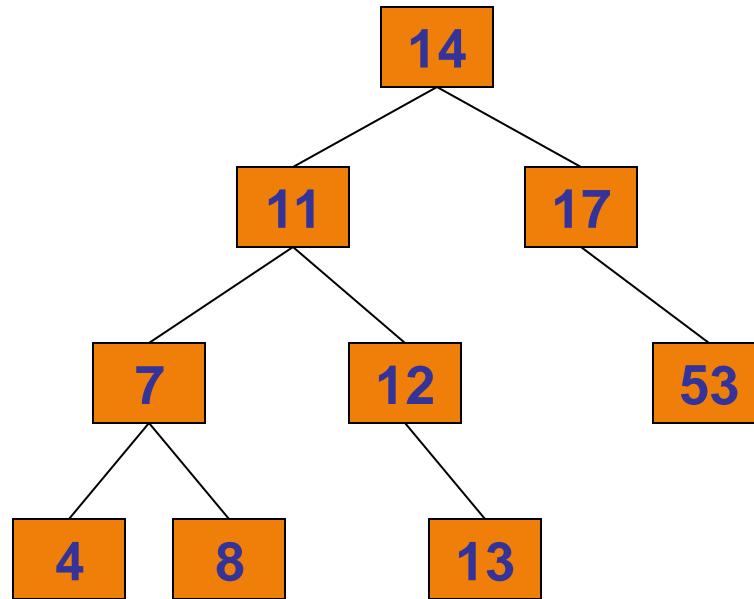
Cây

Bảng băm

Đồ thị

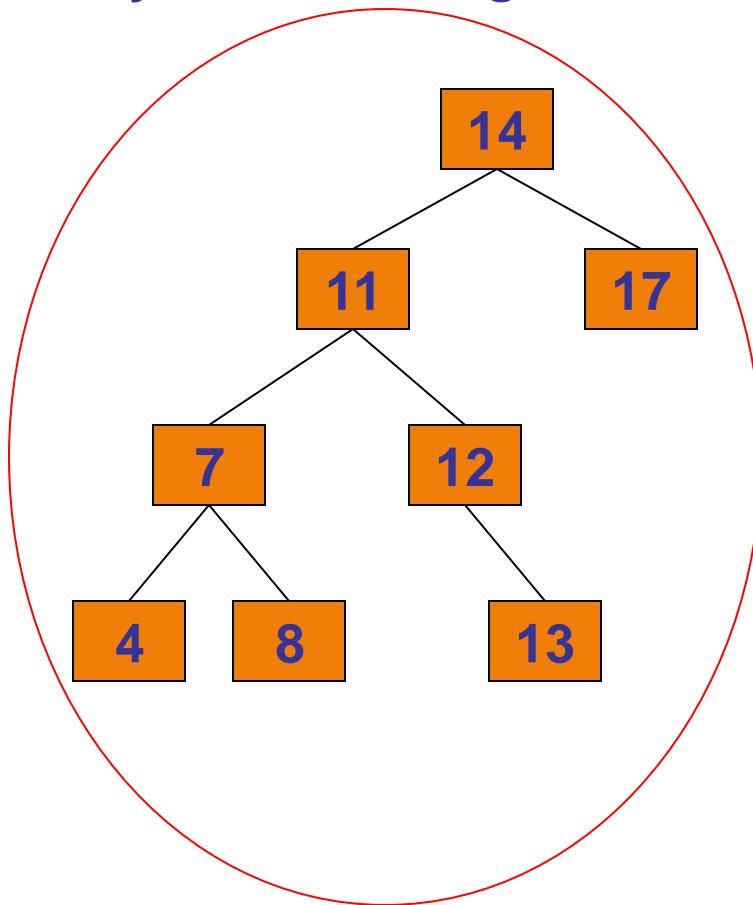
Ví dụ cây AVL:

• Xóa 53



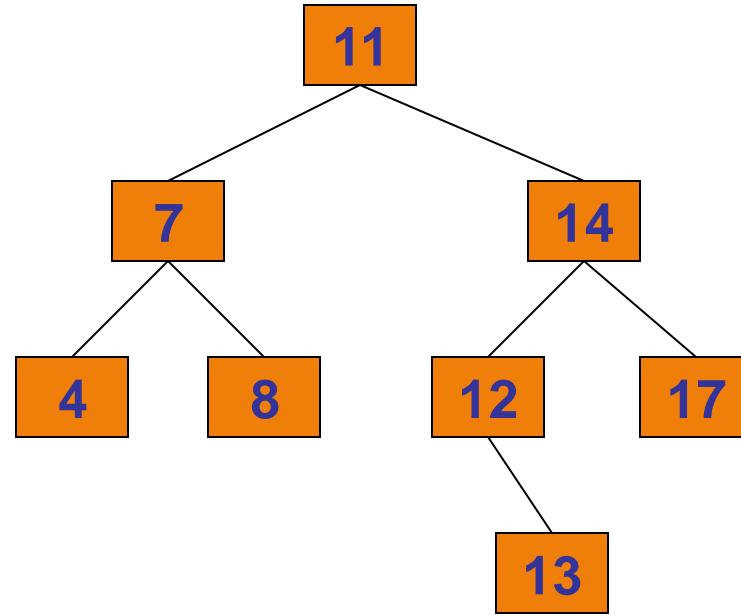
Ví dụ cây AVL:

- Xóa 53, cây mất cân bằng



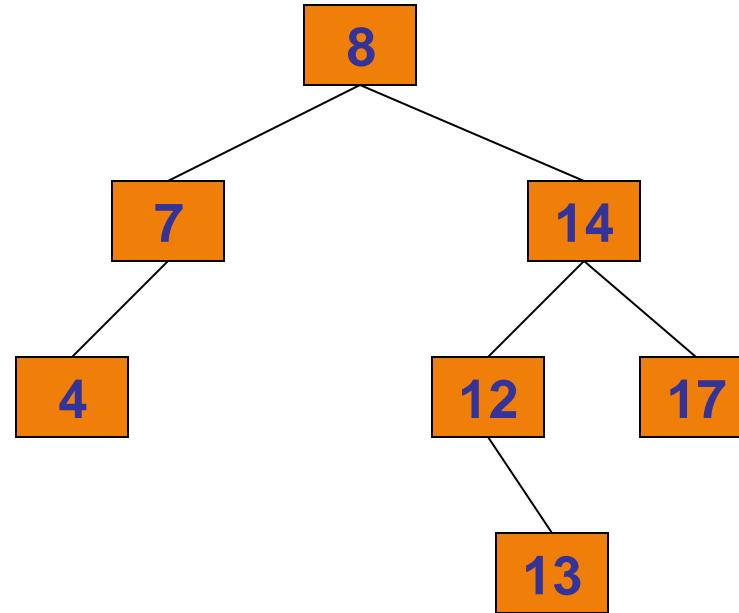
AVL Tree Example:

- Balanced! Remove 11



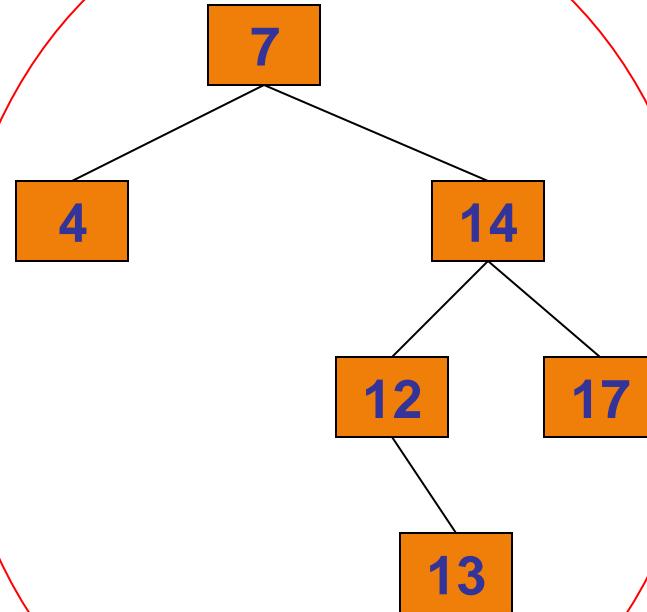
AVL Tree Example:

- Remove 11, replace it with the largest in its left branch



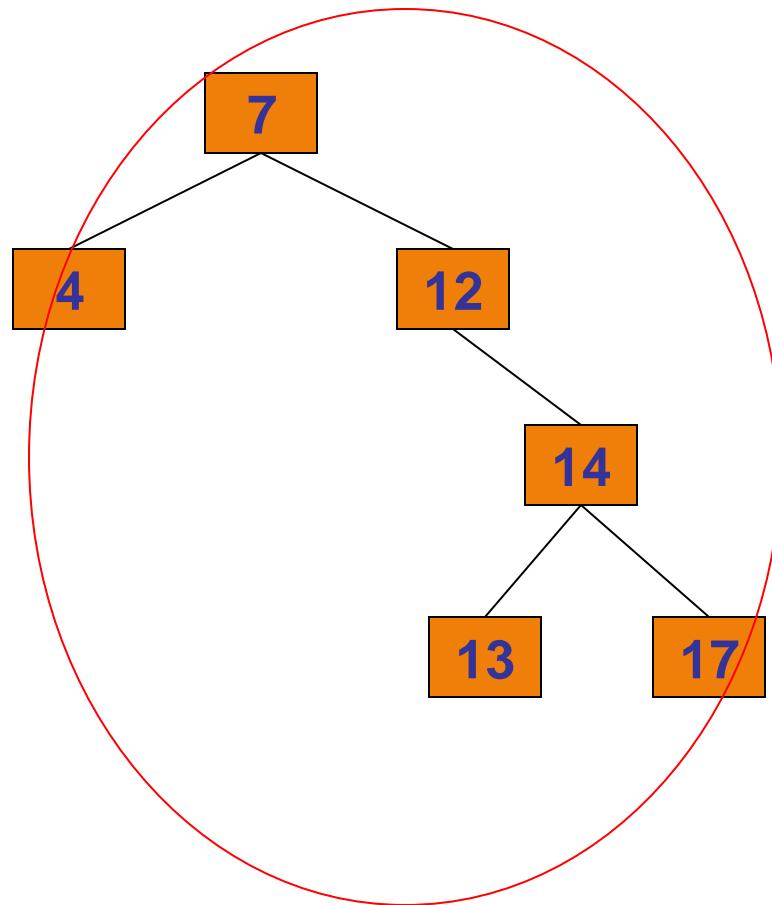
AVL Tree Example:

- Remove 8, unbalanced



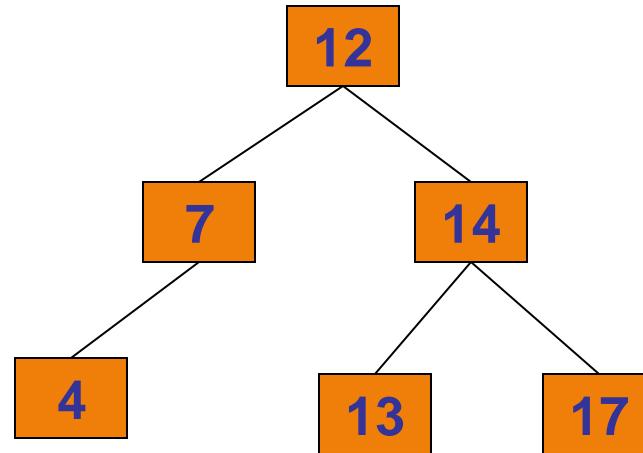
AVL Tree Example:

- Remove 8, unbalanced



AVL Tree Example:

- Balanced!!

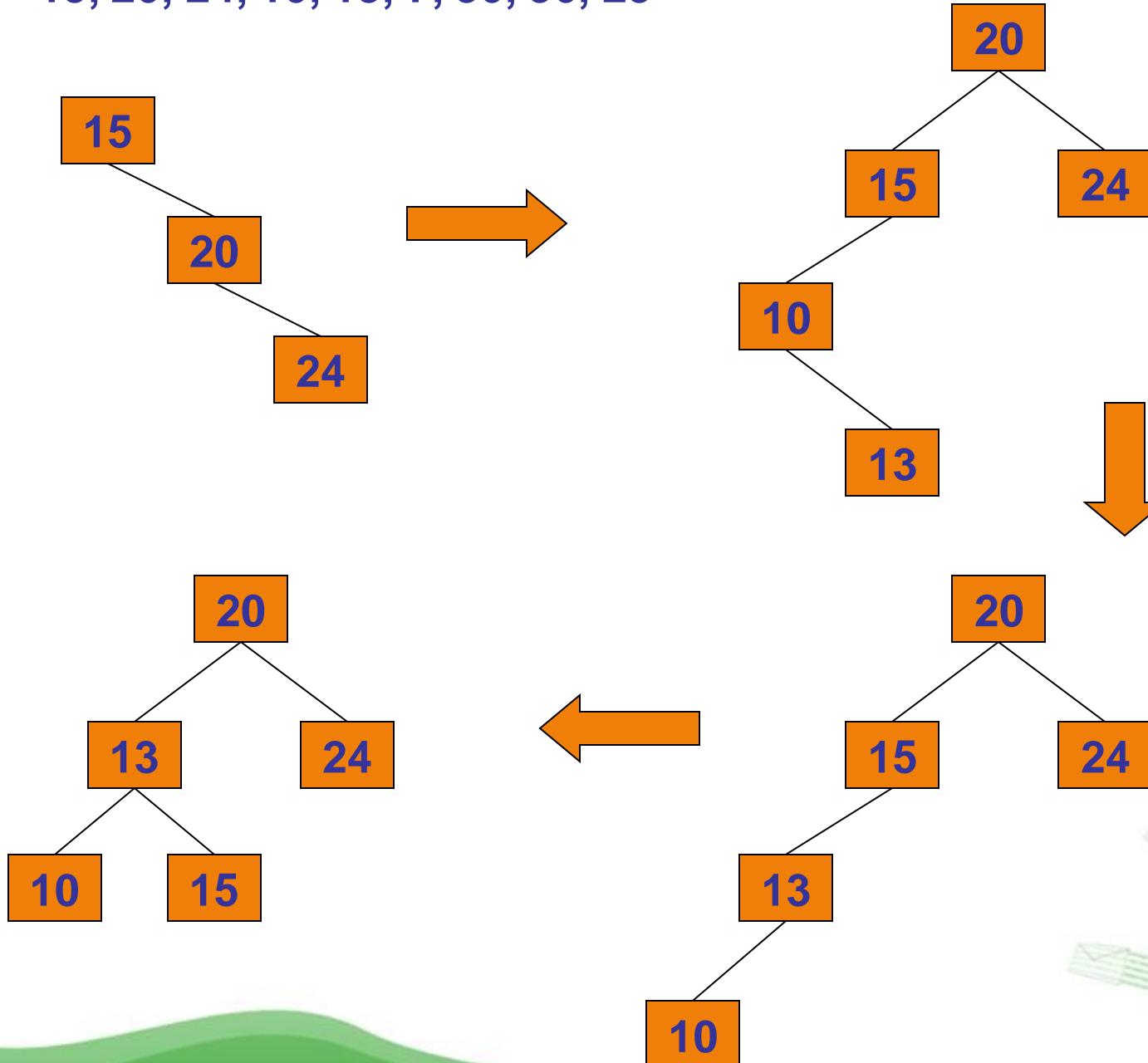


In Class Exercises

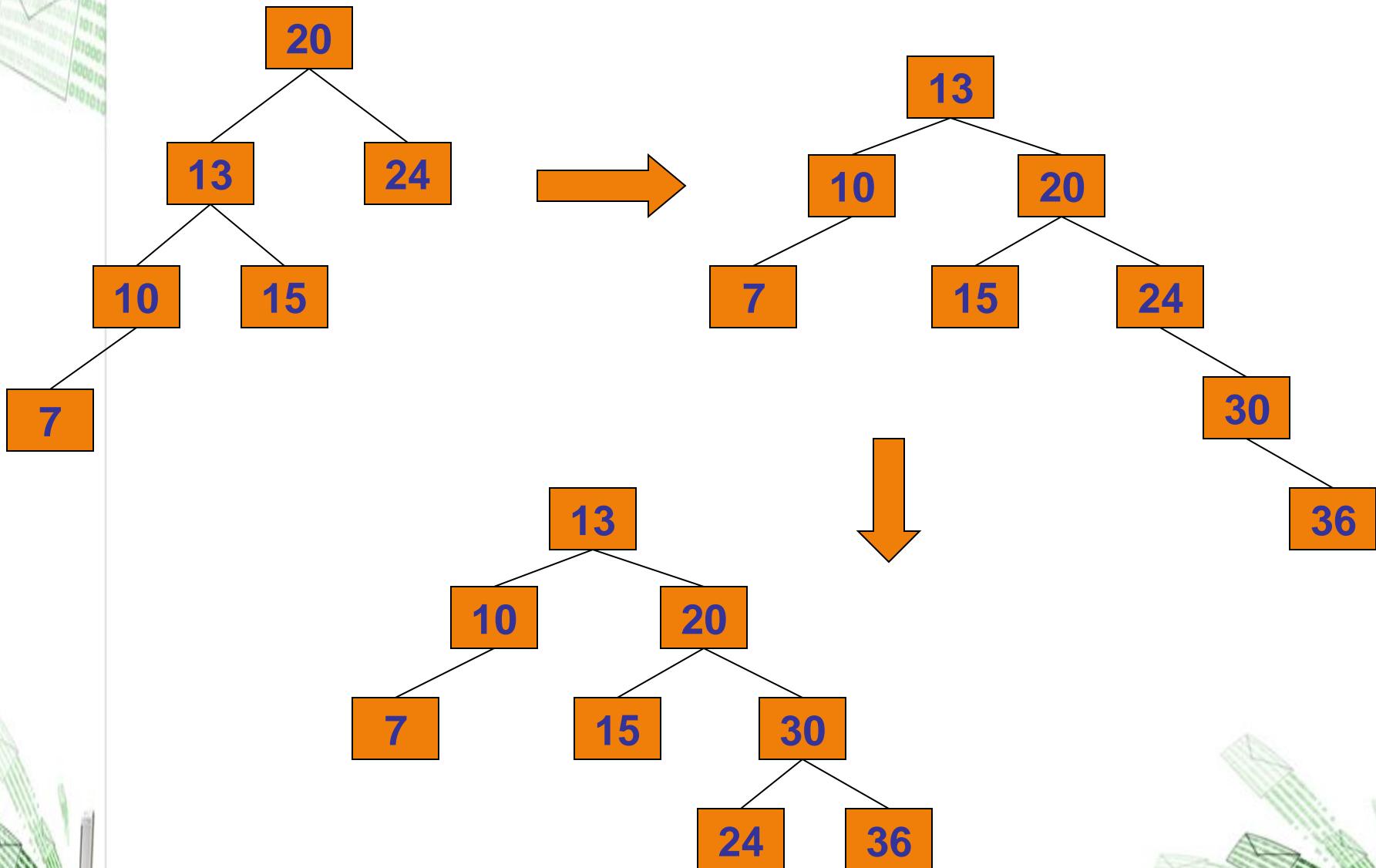
- Lần lượt chèn các giá trị sau vào 1 cây AVL:
15, 20, 24, 10, 13, 7, 30, 36, 25



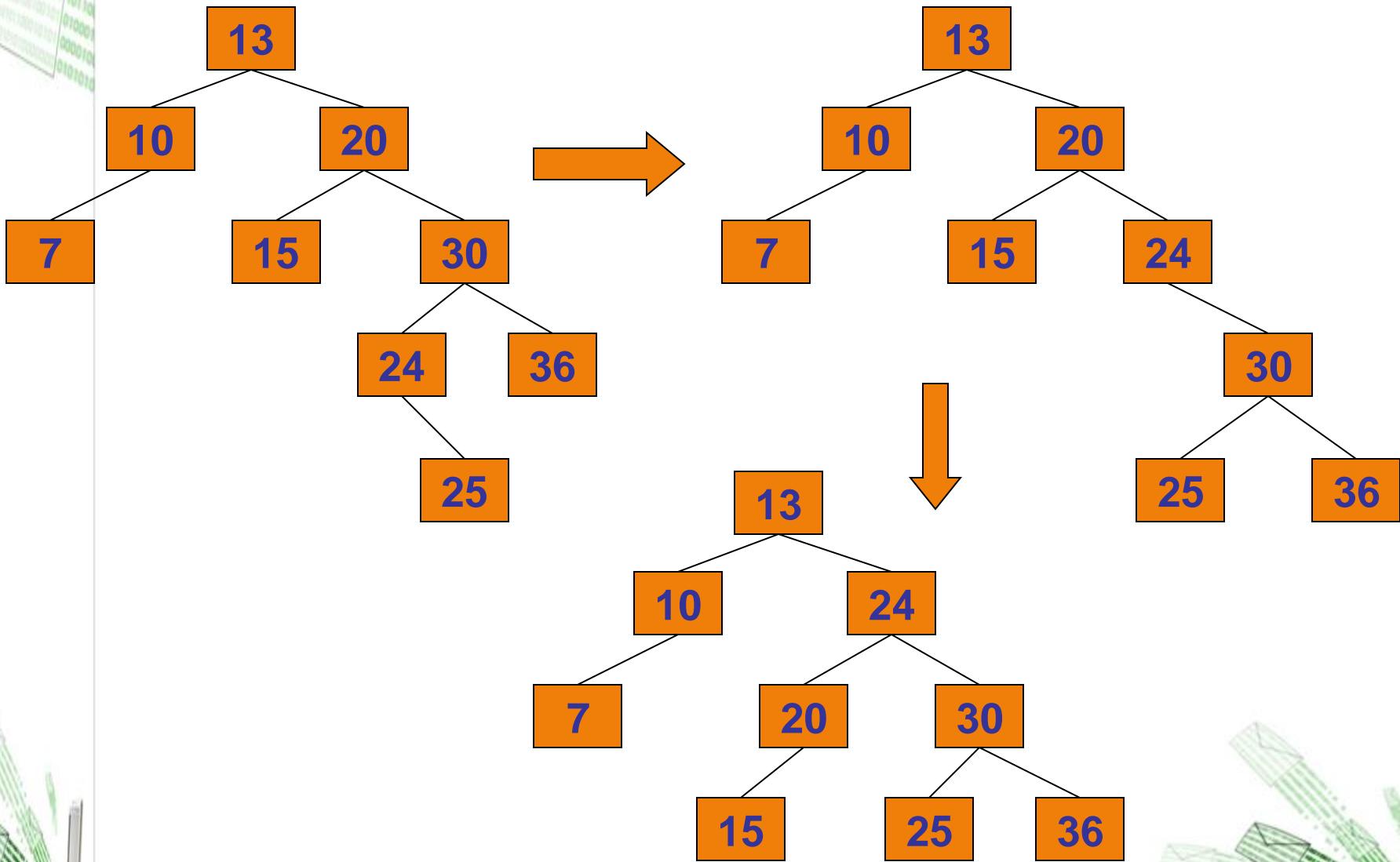
15, 20, 24, 10, 13, 7, 30, 36, 25



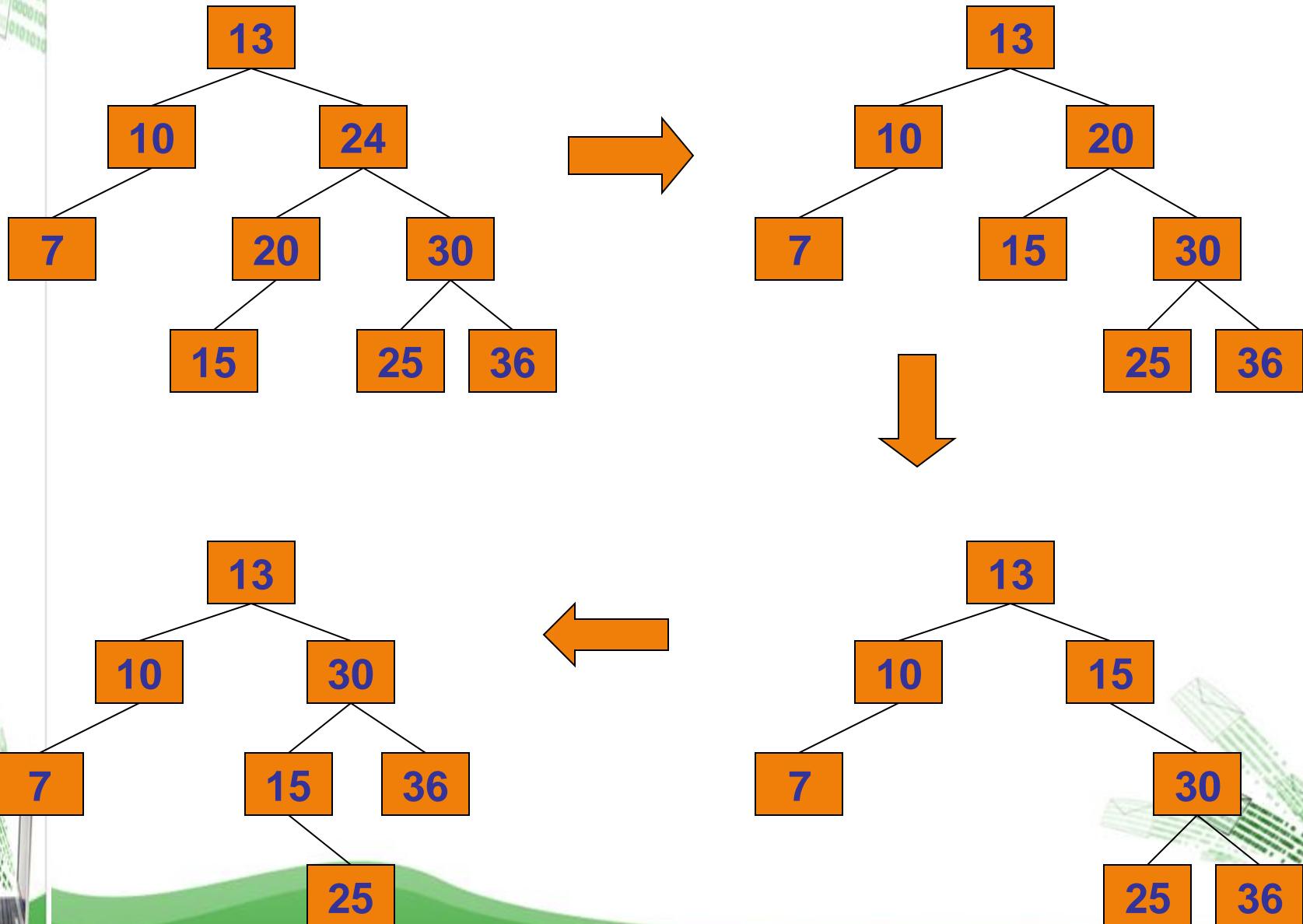
15, 20, 24, 10, 13, 7, 30, 36, 25



15, 20, 24, 10, 13, 7, 30, 36, 25



Remove 24 and 20 from the AVL tree.





Cây

Bảng băm

Đồ thị

Cây tán loe

CÂY TÁN LOE

1. Giới thiệu:

Khi chúng ta truy cập dữ liệu trong một tập dữ liệu thì thông thường chỉ có khoảng 10 phần trăm dữ liệu được truy cập thường xuyên, còn 90 phần trăm dữ liệu còn lại rất ít khi sử dụng đến. Như vậy vấn đề đặt ra là chúng ta cần có cấu trúc dữ liệu sao cho những dữ liệu truy cập thường xuyên được đưa lên “phía trước” để tìm kiếm nhanh hơn còn những dữ liệu ít truy cập thì đưa ra “phía sau”. Cấu trúc dữ liệu cây tán loe được xây dựng nhằm đạt được mục tiêu đó.

Cây

Bảng băm

Đồ thị

CÂY TÁN LOE

2. Khái niệm:

Cây tán loe (Splay Tree) được giới thiệu bởi Sleator và Tarjan vào năm 1985. Cấu trúc Splay Tree thuộc về lớp các cây tìm kiếm nhị phân tự cân bằng (Self-Adjusting Binary Search Tree - BST), có nghĩa là nó sẽ tự điều chỉnh cấu trúc của nó mỗi khi có một thao tác thực hiện trên cây. Mục tiêu của việc điều chỉnh là chuyển những phần tử thường truy cập đưa lên gần node gốc của cây hơn.

Cây

Bảng băm

Đồ thị

CÂY TÁN LOE

3. Ưu nhược điểm:

a. Ưu điểm:

- Cây tán loe có ưu điểm đơn giản, dễ cài đặt hơn cây AVL Tree và cây Red Black Tree.
- Hiệu quả hơn nhiều nếu như phân phôi truy nhập các khóa là không đồng nhất. Cụ thể, nếu một khóa được truy nhập liên tục nhiều lần thì lần truy nhập đầu tiên mất thời gian $O(\log(n))$ còn các lần sau đó là $O(1)$.

b. Nhược điểm:

- các phần tử dữ liệu ít khi sử dụng thì tốc độ truy cập khá chậm (độ phức tạp trong trường hợp tốt nhất là $O(\lg(n))$ và trong trường hợp xấu nhất là $O(n)$).

Cây

Bảng băm

Đồ thị

CÂY TÁN LOE

4. Cấu trúc:

Cấu trúc của cây tán loe giống cấu trúc của cây nhị phân tìm kiếm thông thường.

```
typedef int TYPEINFO;  
typedef struct SPLAYNODE {  
    TYPEINFO info;  
    SPLAYNODE* left;  
    SPLAYNODE* right;  
};  
typedef SPLAYNODE* SPLAYTREE;
```

Cây

Bảng băm

Đồ thị

CÂY TÁN LOE

Cây

Bảng băm

Đồ thị

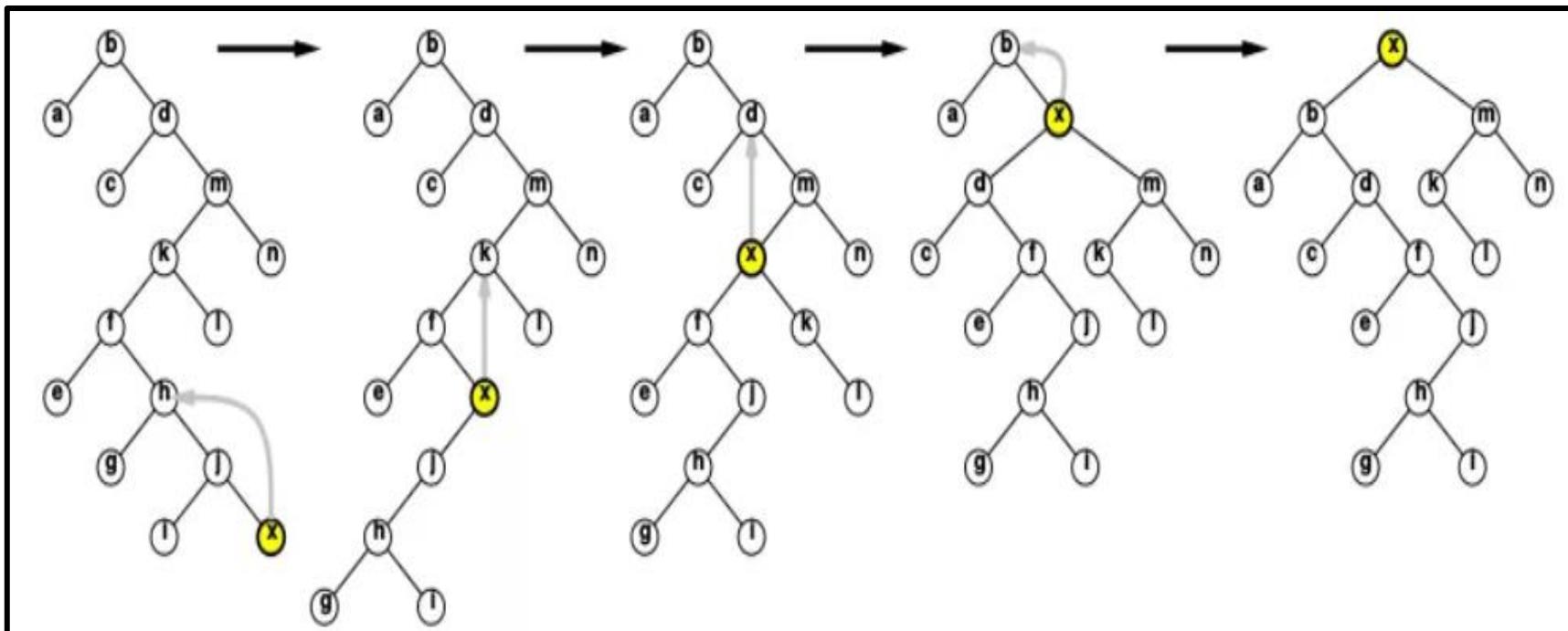
5. Các thao tác trên cây tán loe:

- Phép toán loe cây tại node chứa khóa x.
- Phép toán tìm kiếm một khóa x trên cây
- Phép toán chèn một khóa x vào cây
- Phép toán loại bỏ một khóa x khỏi cây

CÂY TÁN LOE

a. Phép toán loe cây tại node chứa khóa x:

Phép toán này sẽ di chuyển nút x trở thành nút gốc của cây và cây vẫn đảm bảo là cây tìm kiếm nhị phân



CÂY TÁN LOE

Cây

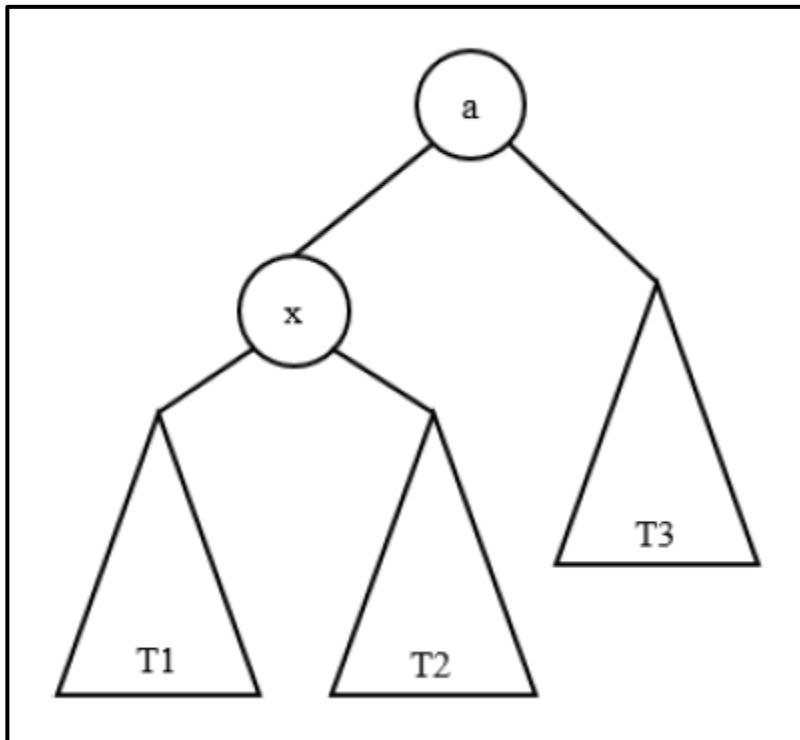
Bảng băm

Đồ thị

- Bản chất một phép toán loe cây tại node x là một chuỗi các phép quay liên tiếp để đưa x trở thành node gốc.
- Tùy vào trường hợp cụ thể mà chúng ta thực hiện các phép quay tương ứng. Chúng ta có các trường hợp cụ thể:
 - x là con node gốc (trái, phải)
 - Zig-Zig (trái, phải)
 - Zig-Zag (trái, phải)

CÂY TÁN LOE

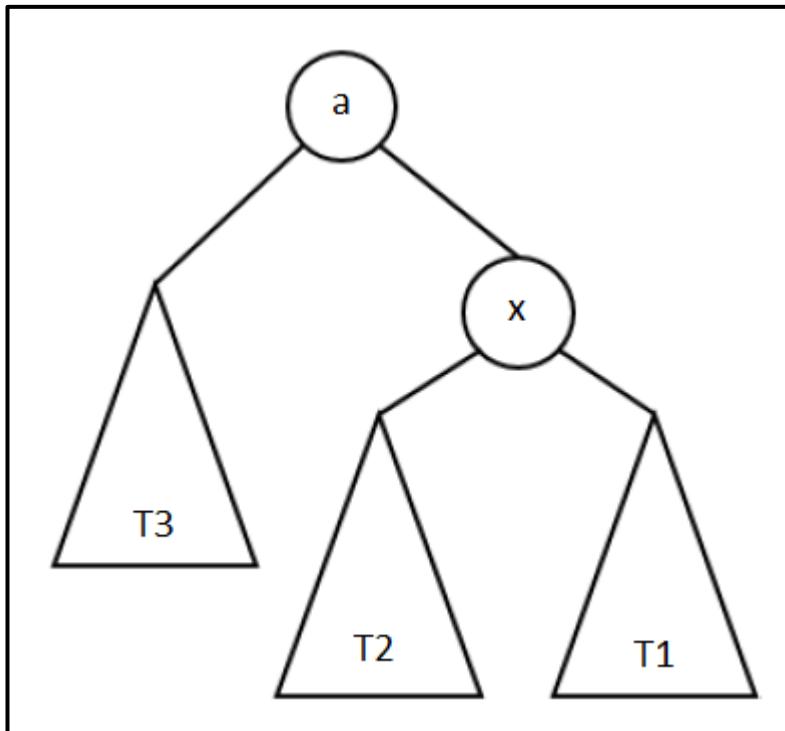
➤ X là con trái node gốc a:



Thực hiện :
Quay phải tại a

CÂY TÁN LOE

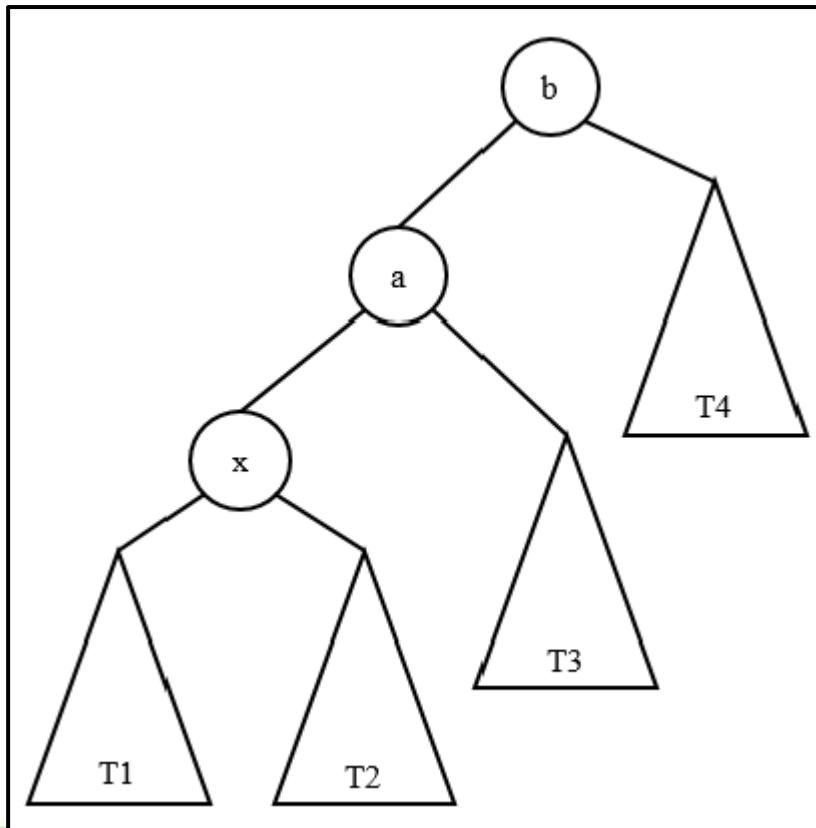
➤ X là con phải node gốc a:



Thực hiện :
Quay trái tại a

CÂY TÁN LOE

- Zig-Zig trái: x là con trái của a, a là con trái của b



Cây

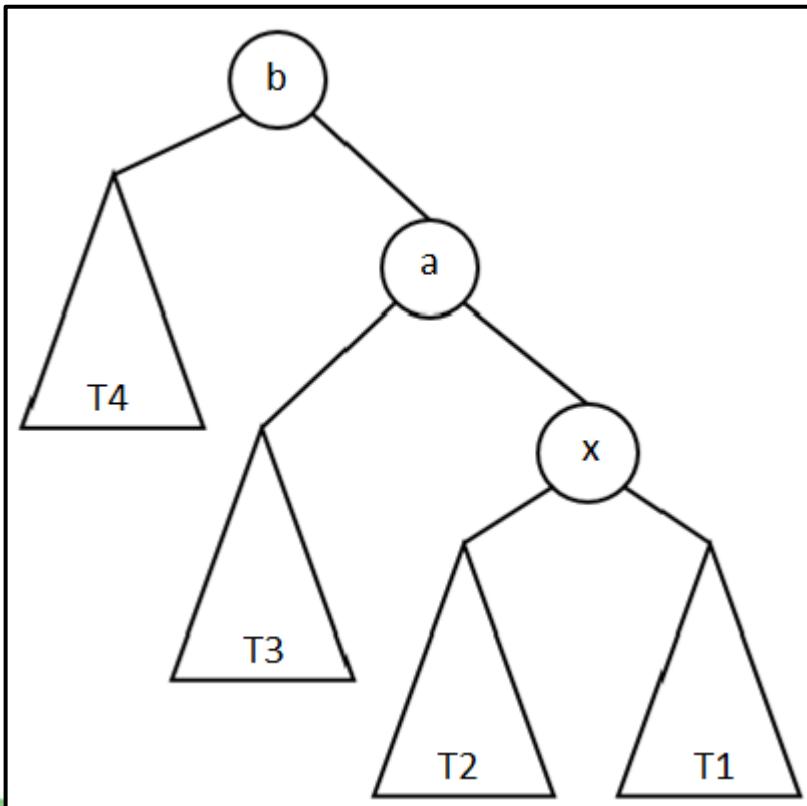
Bảng băm

Đồ thị

Thực hiện liên tiếp:
Quay phải tại b
Quay phải tại a

CÂY TÁN LOE

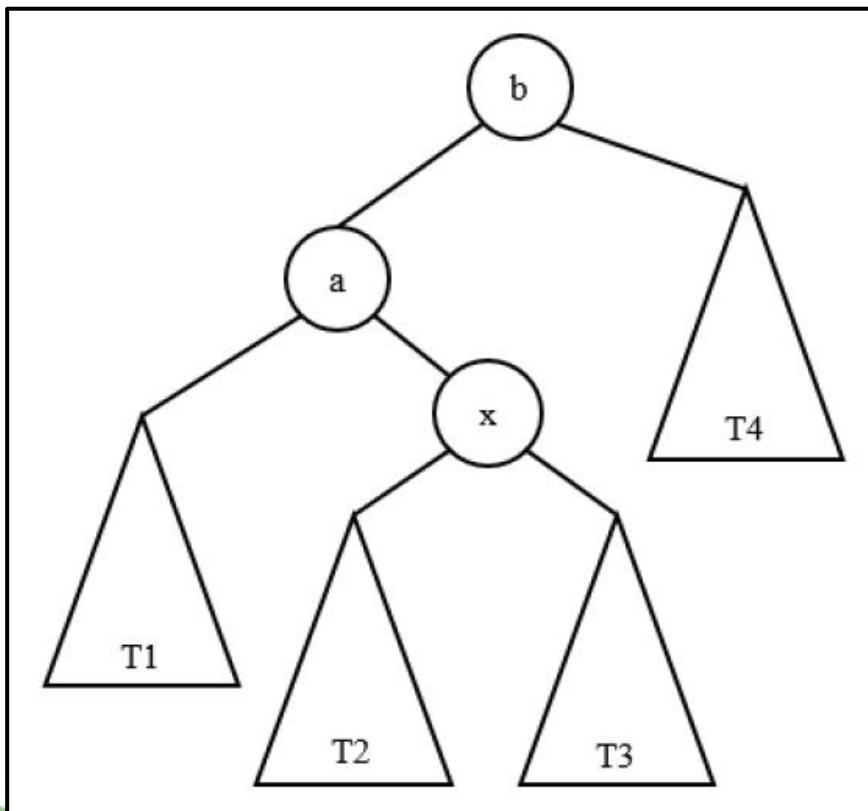
- Zig-Zig phải: x là con phải của a, a là con phải của b



Thực hiện liên tiếp:
Quay trái tại b
Quay trái tại a

CÂY TÁN LOE

- Zig-Zag trái: x là con phải của a, a là con trái của b



Cây

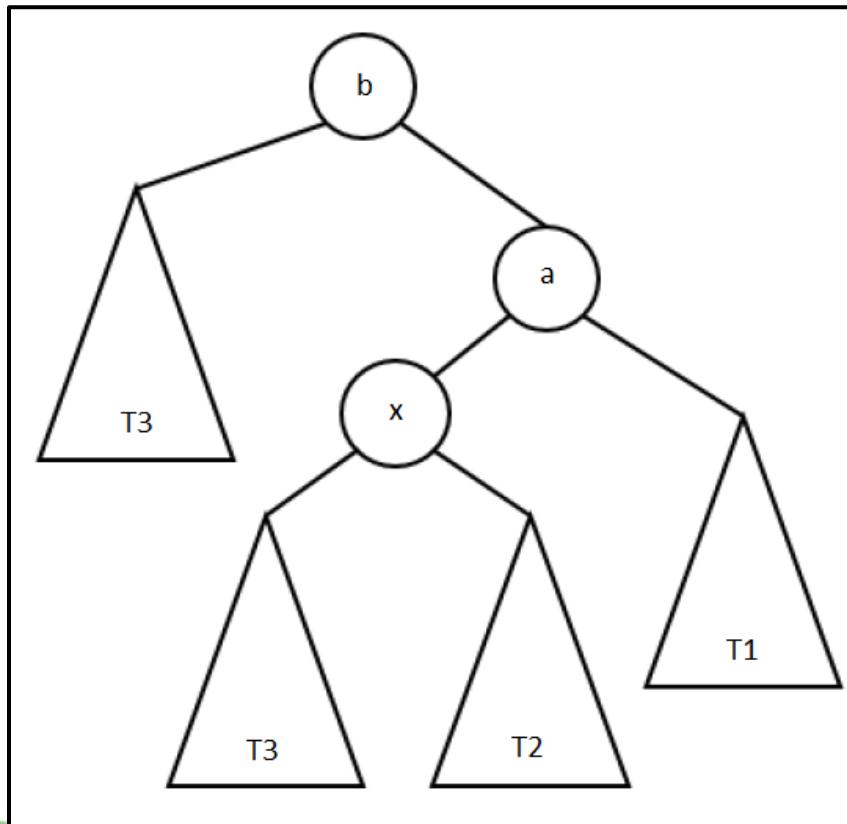
Bảng băm

Đồ thị

Thực hiện liên tiếp:
Quay trái tại a
Quay phải tại b

CÂY TÁN LOE

- Zig-Zag phải: x là con trái của a, a là con phải của b



Cây

Bảng băm

Đồ thị

Thực hiện liên tiếp:

Quay phải tại a

Quay trái tại b

Thuật toán splay cây tại node x

void splay(SPLAYTREE &S, TYPEINFO k)

Gọi x là node chứa khóa k.

B1: nếu x là node gốc thoát ngược lại qua B2

B2: nếu x là con trái (phải) của node gốc

- Xoay phải (trái) tại node gốc ngược lại qua B3

B3: lúc này chắc chắn có cha a và ông b

- kiểm tra 4 trường hợp zig-zig và zig-zag và thực hiện các phép quay tương ứng

- Đệ quy loe tại node x mới

Cây

Bảng băm

Đồ thị

CÂY TÁN LOE

Cây

Bảng băm

Đồ thị

b. Phép toán tìm kiếm một khóa k trên cây:

Việc tìm kiếm thực hiện tương tự như tìm kiếm trên cây BST. Tuy nhiên chúng ta cần thực hiện phép loe cây sau khi tìm kiếm.

- Nếu tìm thấy và x là nút có khóa k thì làm loe cây tại nút x.
- Nếu không tìm thấy và quá trình tìm kiếm dừng tại nút x thì làm loe cây tại nút x.

CÂY TÁN LOE

Cây

Bảng băm

Đồ thị

c. Phép toán chèn một khóa k vào cây:

Việc chèn khóa k vào cây kết quả là thêm một node x có khóa k vào cây. Sau khi việc chèn kết thúc chúng ta cần làm loe cây tại nút x.

Thuật toán chèn

```
void Insert(SPLAYTREE &S, TYPEINFO k)
```

B1: nếu S rỗng => S là node mới chứa khóa k
ngược lại qua B2

B2: nếu S chứa khóa k => thoát ngược lại qua B3

B3: nếu S chứa khóa lớn hơn k

- Chèn k vào cây con trái của S
- Loại cây S tại node con trái của S

Ngược lại:

- Chèn k vào cây con phải của S
- Loại cây S tại node con phải của S

Cây

Bảng băm

Đồ thị

CÂY TÁN LOE

Cây

Bảng băm

Đồ thị

d. Phép toán loại bỏ một khóa k khỏi cây T:

B1: Thực hiện phép toán tìm khóa k trong cây. Nếu tìm thấy, khi đó x là nút gốc của cây T. Gọi TL là cây con trái và TR là cây con phải của cây T.

B2: Nếu x chỉ có 1 con thì cây T mới chính là cây con của x, xóa node x. Ngược lại tìm y là nút tận cùng bên phải trong cây TL. Loại cây TL tại nút y. Khi đó TL có nút gốc là nút y và cây con phải của y bằng null.

B3: Gán cây TR vào cây con phải của cây TL (TL chính là y) và loại bỏ nút x.

CÂY TÁN LOE

Cây**Bảng băm****Đồ thị**

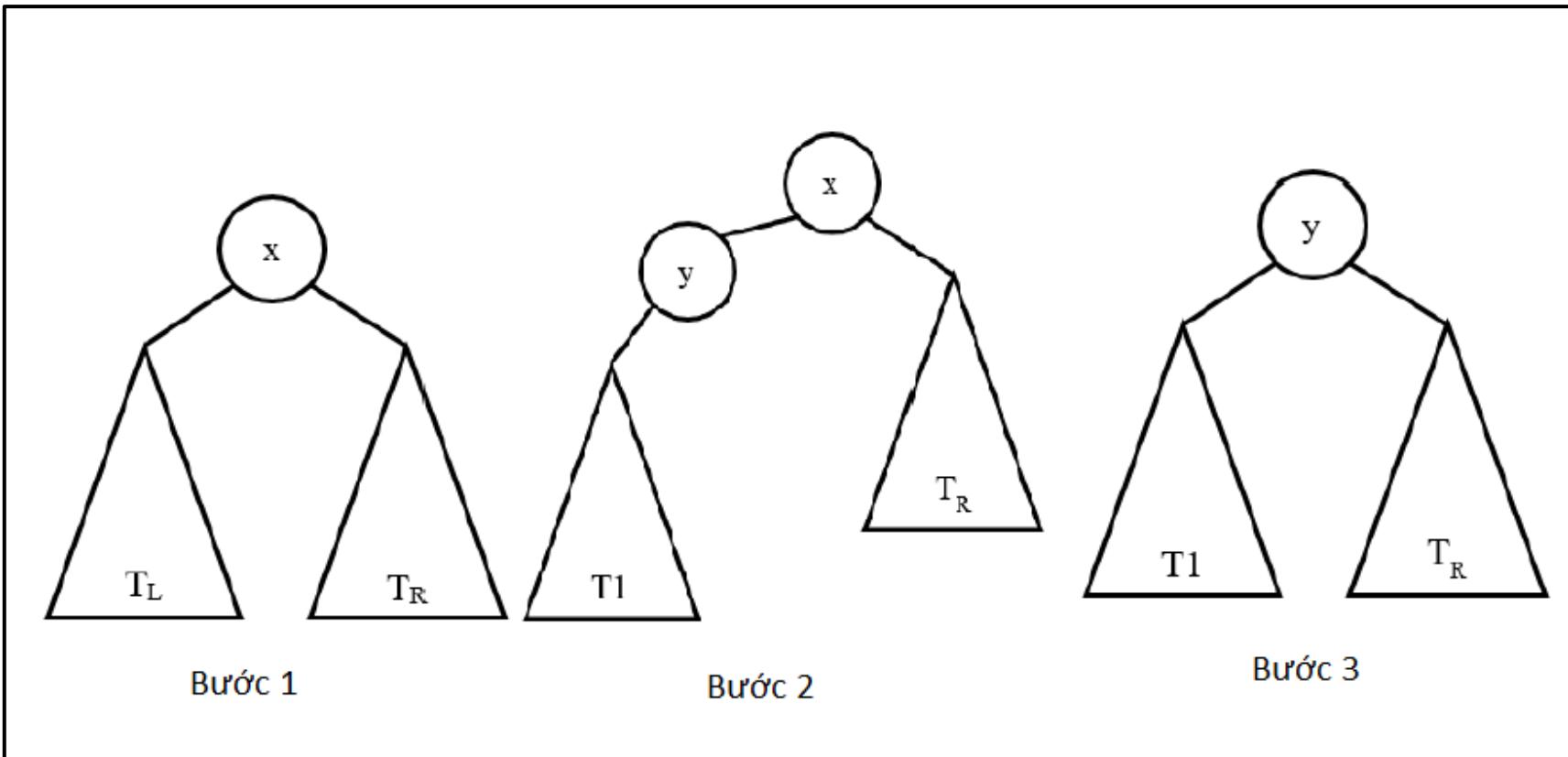
Delete 53, 11, 8

CÂY TÁN LOE

Cây

Bảng băm

Đồ thị



Cây

Bảng băm

Đồ thị

Hàng ưu tiên

HÀNG ƯU TIÊN

1. Giới thiệu: Hàng đợi là một danh sách hoạt động theo cơ chế là “vào trước thì ra trước”. Trong thực tế có nhiều công việc hoạt động theo cơ chế này nhưng có bổ sung thêm chế độ ưu tiên. Ví dụ như trong **Bệnh Viện**, tại **cửa an ninh sân bay (ưu tiên gì?)**.

Như vậy để công việc hoạt động được theo chế độ ưu tiên này thì cần phải xây dựng cấu trúc dữ liệu hàng đợi có kèm theo độ ưu tiên. Những phần tử có độ ưu tiên cao được đưa lên phía đầu hàng đợi và những phần tử có cùng độ ưu tiên thì vẫn hoạt động theo cơ chế của hàng đợi.

Cây

Bảng băm

Đồ thị

HÀNG ƯU TIÊN

2. Khái niệm:

Hàng ưu tiên là hàng đợi mà mỗi phần tử có độ ưu tiên. Những phần tử có cùng độ ưu tiên thì hoạt động theo cơ chế hàng đợi, còn những phần tử có độ ưu tiên khác nhau thì phần tử có độ ưu tiên cao được đưa lên phía đầu hàng đợi. Ví dụ phần tử A có độ ưu tiên 2 được ký hiệu (A, 2), và cho hàng ưu tiên như sau: {(B, 1), (A, 2), (C, 3)}, khi đó nếu thêm phần tử (D, 2) vào hàng ưu tiên thì có được: {(B, 1), (A, 2), (D, 2), (C, 3)}.

Cây

Bảng băm

Đồ thị

HÀNG ƯU TIÊN

3. Các phép toán:

- Chèn phần tử: Vị trí chèn là đứng trước các phần tử có độ ưu tiên thấp hơn nó và đứng sau các phần tử có cùng độ ưu tiên và các phần tử có độ ưu tiên cao hơn.
- Loại bỏ phần tử: Phần tử bị loại bỏ là phần tử đứng đầu hàng ưu tiên.
- Trả về phần tử: Phần tử được trả về là phần tử đứng đầu hàng ưu tiên.

Cây

Bảng băm

Đồ thị

HÀNG ƯU TIÊN

4. Cài đặt:

- Hàng ưu tiên có thể được biểu diễn bằng nhiều cấu trúc như danh sách đặc, danh sách liên kết, cây thứ tự bộ phận, cây nghiêng, heap,...
- Trong nội dung chương trình chúng ta sẽ tìm hiểu cách cài đặt và biểu diễn của hàng ưu tiên bằng **cây thứ tự bộ phận** và **cây nghiêng**.

Cây

Bảng băm

Đồ thị



Cây

Bảng băm

Đồ thị

Cây thứ tự bộ phận (partially ordered trees)

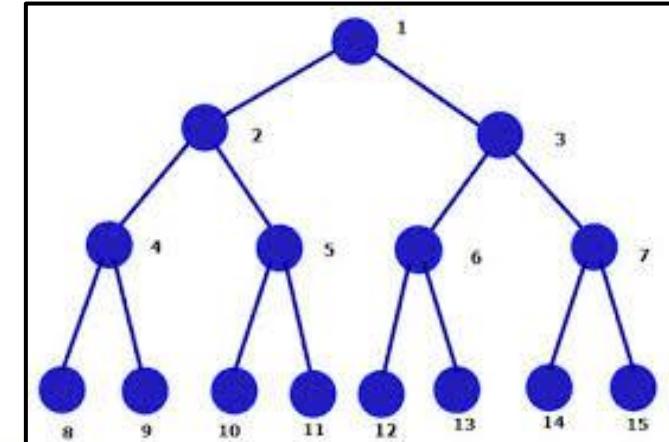
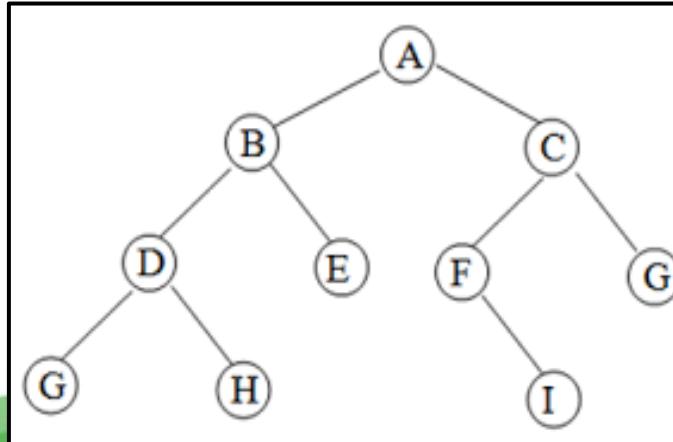
CÂY THÚ TỰ BỘ PHẬN

1. Khái niệm:

➤ **Cây nhị phân hoàn hảo (Perfect Binary Tree):**

Cây nhị phân hoàn hảo chiều cao h là 1 cây nhị phân thỏa:

- Tất cả các node lá đều có cùng độ sâu h.
- Tất cả các node không là node lá đều có đủ 2 con.



CÂY THÚ TỰ BỘ PHẬN

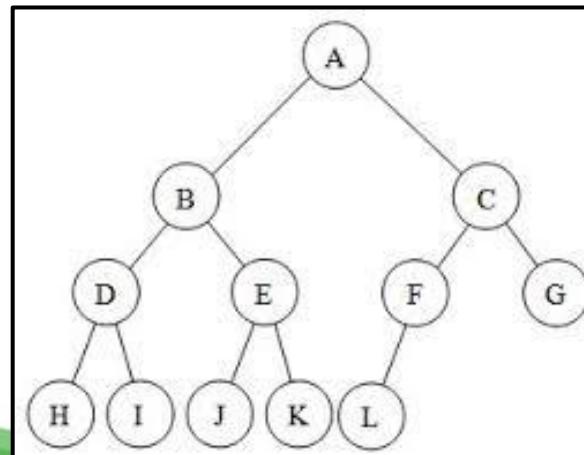
Cây

Bảng băm

Đồ thị

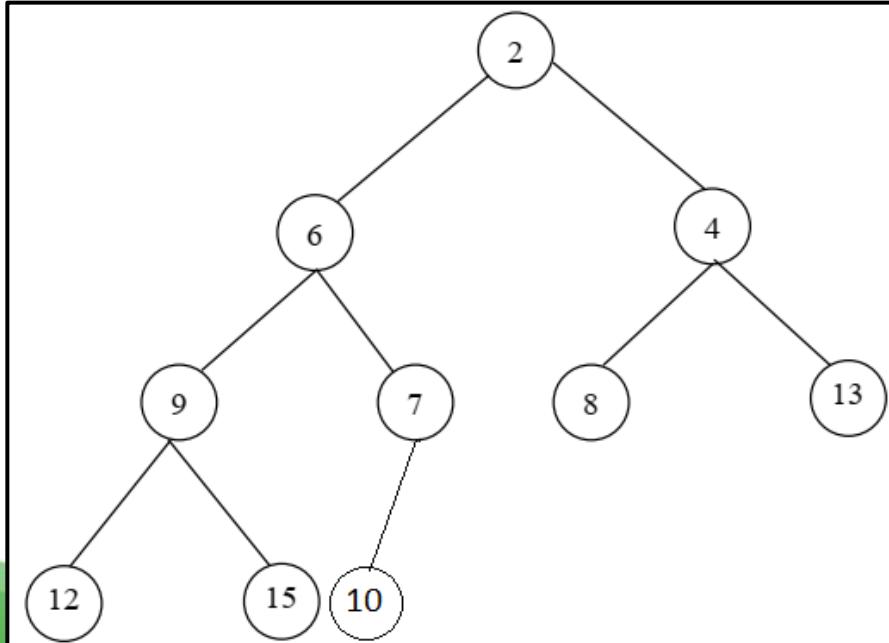
➤ **Cây nhị phân hoàn toàn (Complete Binary Tree):** là cây nhị phân thỏa mãn:

- Cây nhị phân có độ cao $h=0$
- Nếu cây có độ cao $h \geq 1$ thì kể từ mức $(h-1)$ trở lên là cây nhị phân đầy đủ, và ở mức h nếu với mọi nút có một nút con thì các nút đứng trước nó (nếu có) có đầy đủ hai con, nếu một nút có một nút con thì phải là nút con trái.



CÂY THÚ TỰ BỘ PHẬN

- **Cây thứ tự bộ phận (Partially Ordered Tree):** là cây nhị phân hoàn toàn sao cho với mọi nút của cây thì khóa của nút đó nhỏ hơn hoặc bằng khóa của hai nút con (nếu có).



Cây

Bảng băm

Đồ thị

CÂY THÚ TỰ BỘ PHẬN

2. Các phép toán:

- Chèn phần tử:
- Loại bỏ phần tử ở gốc của cây (lưu ý đây là node có khóa nhỏ nhất trong cây).

Cây

Bảng băm

Đồ thị

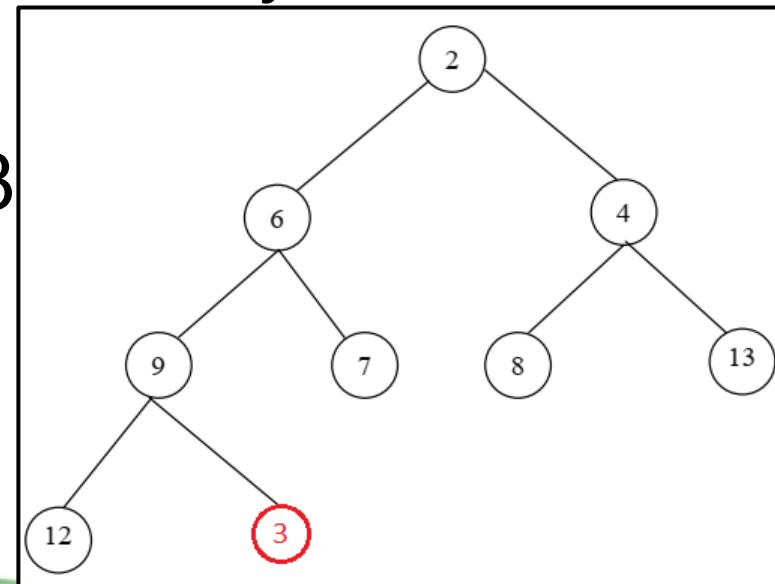
CÂY THỨ TỰ BỘ PHẬN

a. Phép toán chèn:

Nút được chèn vào tại mức lớn nhất và nằm ở vị trí tận cùng bên phải. Khi đó cây có thể không còn là Cây thứ tự bộ phận nên phải đưa ra phương pháp hiệu chỉnh cây.

Ví dụ:

sau khi chèn khóa 3
vào thì cây không
còn là cây thứ tự
bộ phận



Cây

Bảng băm

Đồ thị

CÂY THÚ TỰ BỘ PHẬN

Giải thuật hiệu chỉnh cây thứ tự bộ phận cho phép toán chèn:

Giả sử x là nút vi phạm và a là nút cha của nó.

Bước 1:

Nếu khóa của nút a nhỏ hơn hoặc bằng khóa của nút x thì kết thúc giải thuật, ngược lại qua Bước 2.

Bước 2:

Hoán vị giá trị hai nút a và x.

Gán x là nút a và a là nút cha của nó.

Bước 3: Lặp lại Bước 1.

Cây

Bảng băm

Đồ thị

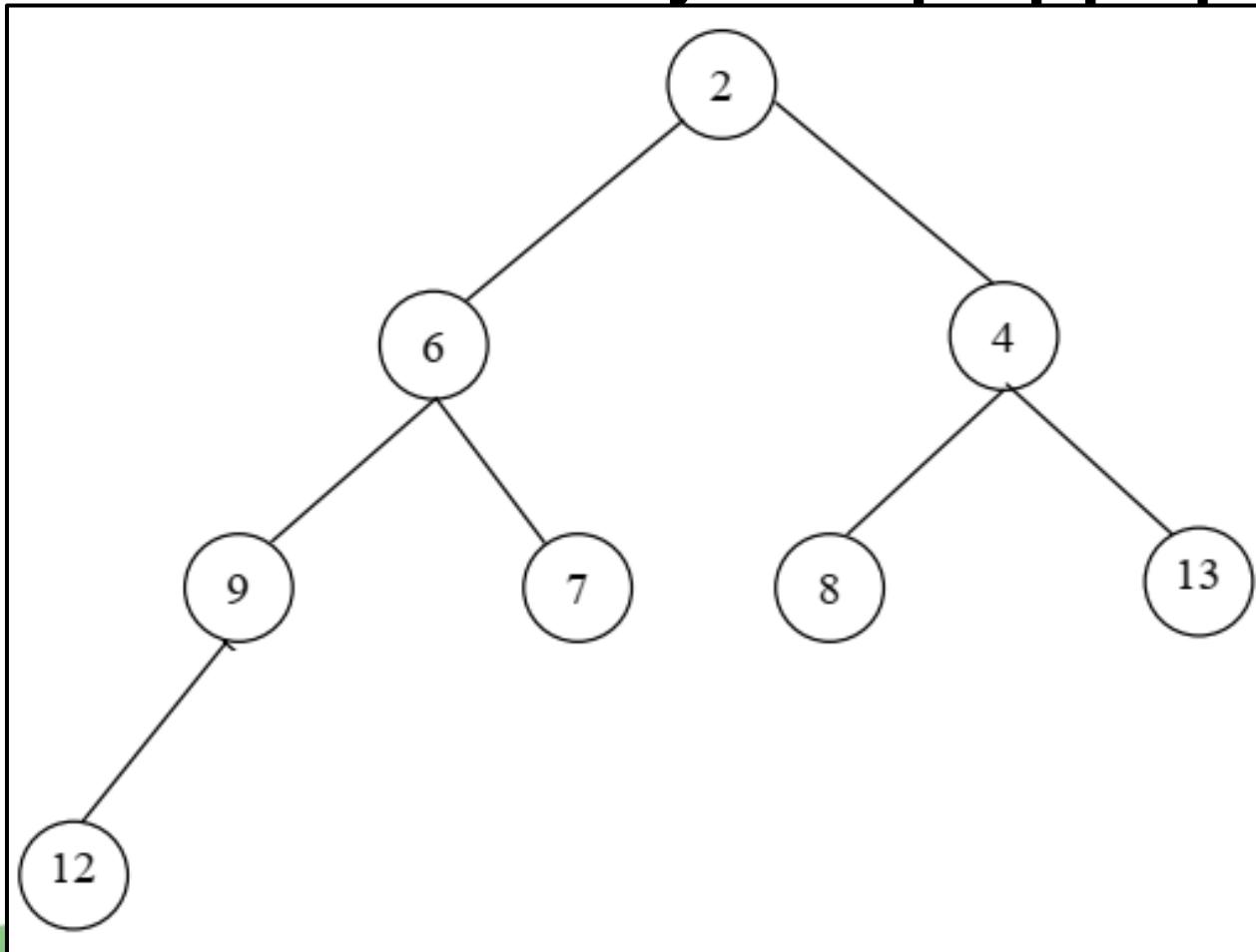
CÂY THÚ TỰ BỘ PHẬN

Cây

Bảng băm

Đồ thị

Demo thêm 3 vào cây thứ tự bộ phận sau :



CÂY THÚ TỰ BỘ PHẬN

b. Phép toán xóa node gốc:

Bước 1: Tìm nút “thế mạng” a là nút tận cùng bên phải có mức lớn nhất. Sau đó sao chép giá trị nút a sang nút gốc và xóa nút a.

Bước 2: Hiệu chỉnh cây, Giả sử x là nút cần hiệu chỉnh.

Bước 2.1: Gọi a là nút có khóa nhỏ nhất trong hai nút con của nút x. Nếu khóa của nút $x \leq$ khóa của nút a thì kết thúc giải thuật, ngược lại qua Bước 2.2.

Bước 2.2: Hoán vị giá trị của hai nút a và x.
Gán nút x bằng nút a.

Bước 2.3: Lặp lại Bước 2.1.

Cây

Bảng băm

Đồ thị

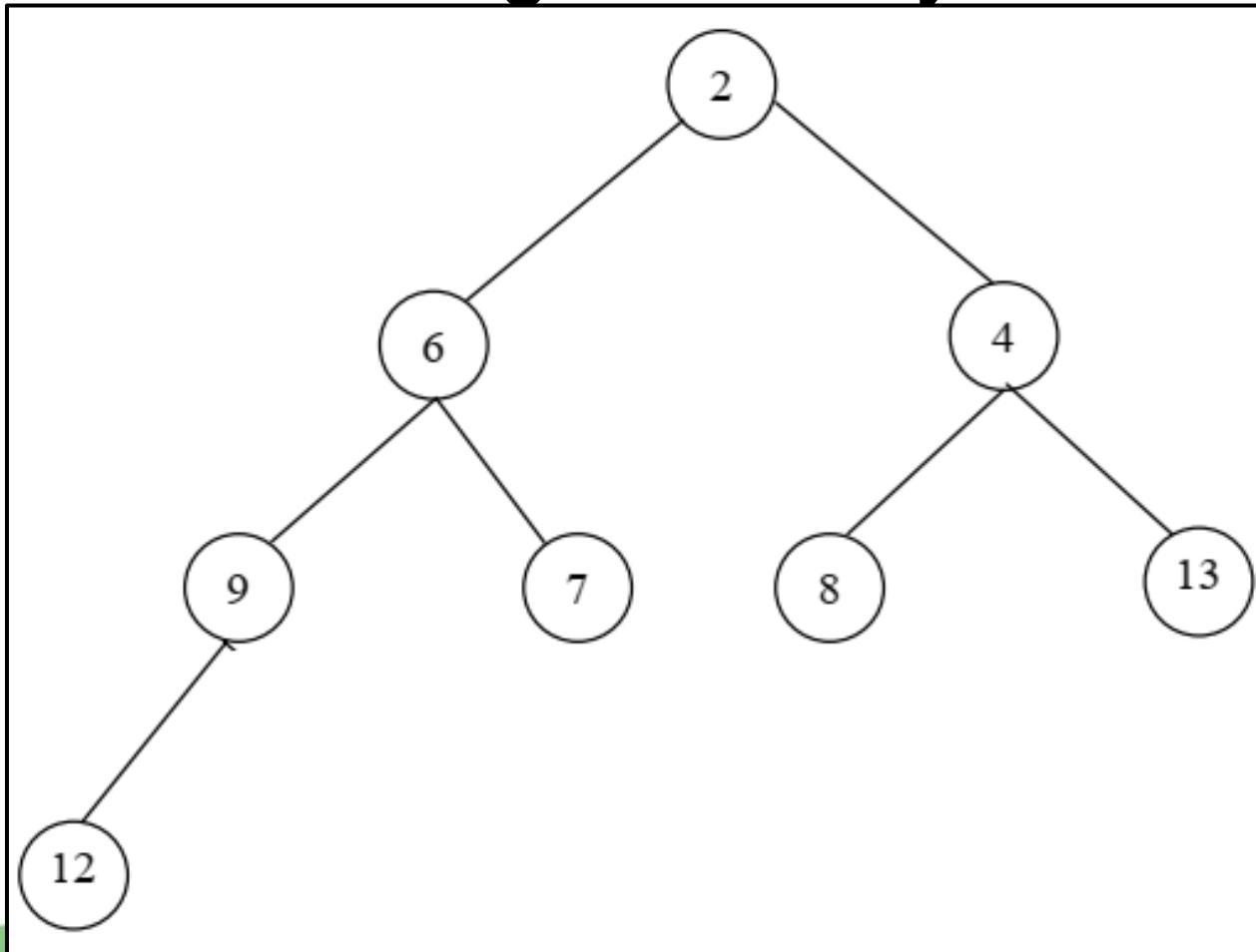
CÂY THÚ TỰ BỘ PHẬN

Cây

Bảng băm

Đồ thị

Demo xóa node gốc của cây sau:



CÂY THÚ TỰ BỘ PHẬN

3. Cài đặt:

Chúng ta có thể sử dụng Cây thứ tự bộ phận để cài đặt Hàng ưu tiên. Bởi vì: Nút gốc của Cây thứ tự bộ phận có khóa nhỏ nhất tương ứng với phần tử có độ ưu tiên cao nhất. Do đó khi Loại bỏ nút có khóa nhỏ nhất trong Cây thứ tự bộ phận tương ứng với phép toán Loại bỏ phần tử khỏi Hàng ưu tiên. Khi chèn một phần tử vào Cây thứ tự bộ phận tương ứng với thao tác chèn một phần tử vào Hàng ưu tiên.

Cây

Bảng băm

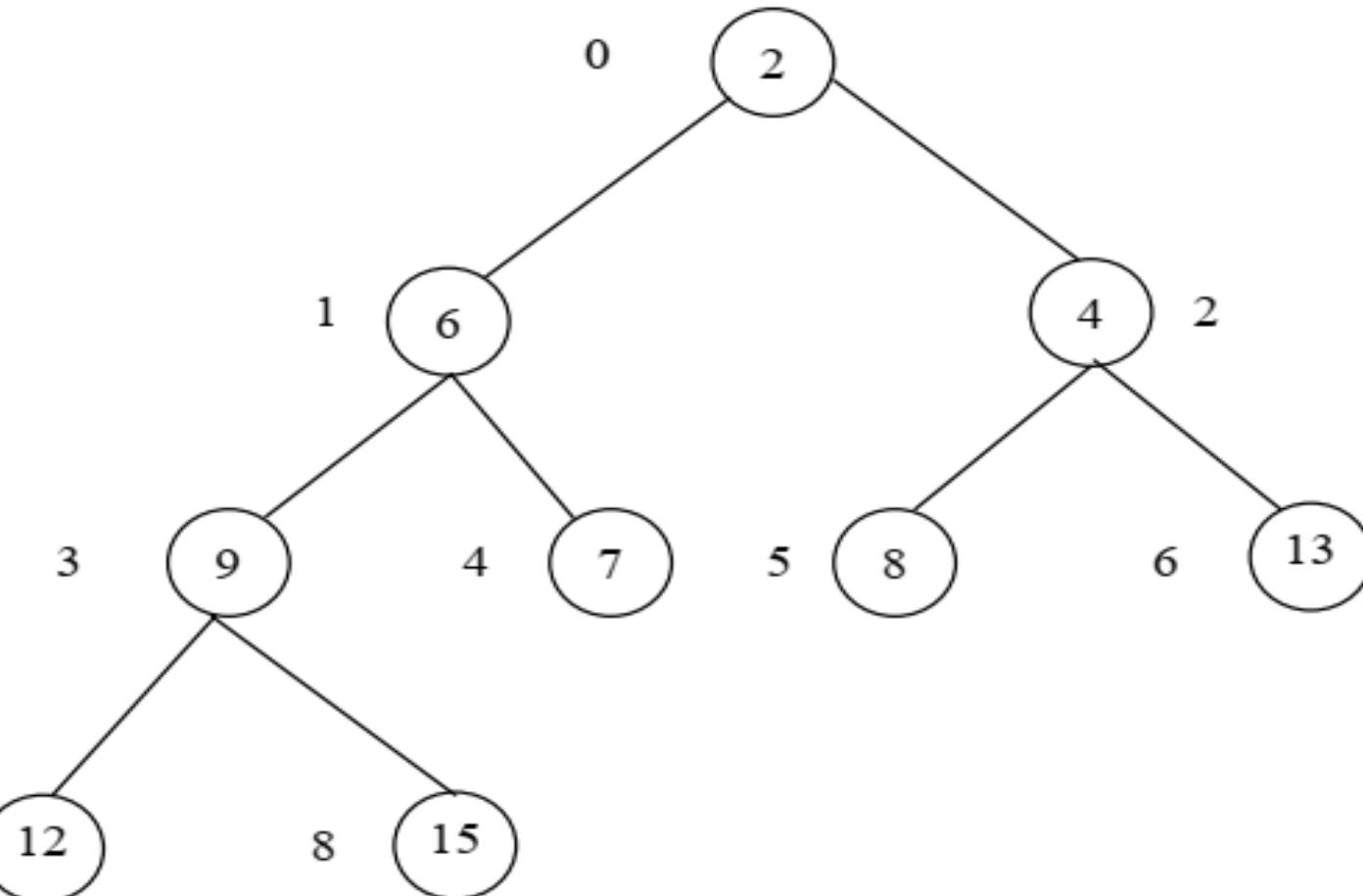
Đồ thị

CÂY THÚ TỰ BỘ PHẬN

Cây

Bảng băm

Đồ thị



CÂY THÚ TỰ BỘ PHẬN

Cây thứ tự bộ phận có thể được cài đặt bằng mảng một chiều, với phương pháp cài đặt là ứng với mỗi nút của Cây thứ tự bộ phận được lưu trữ tại phần tử có chỉ số i của mảng một chiều thì hai nút con của nó (nếu có) được lưu trữ tại các phần tử có chỉ số $(2*i+1)$ và $(2*i+2)$ của mảng.

Cây

Bảng băm

Đồ thị

CÂY THÚ TỰ BỘ PHẬN

4. Cấu trúc:

Cấu trúc dữ liệu cho Hàng ưu tiên:

```
const int MAX = 100;
typedef int TYPEINFO;
struct HangUuTien
{
    TYPEINFO list[MAX];
    int count;
};
```

Cây

Bảng băm

Đồ thị

HÀNG ƯU TIÊN

Các thao tác trên hàng ưu tiên:

+ Phép toán khởi tạo:

void khoitao(HangUuTien& x)

+ Phép toán kiểm tra Hàng ưu tiên rỗng:

int kiemtraRong(HangUuTien x)

+ Phép toán kiểm tra Hàng ưu tiên đầy:

int kiemtraDay(HangUuTien x)

+ Phép toán chèn một phần tử vào Hàng ưu tiên:

void chen(TYPEINFO a, HangUuTien& x)

+ Phép toán tham chiếu đến phần tử đầu tiên:

int getPhantuDautien(HangUuTien x, TYPEINFO& a)

+ Phép toán loại bỏ một phần tử khỏi Hàng ưu tiên:

void xoa(HangUuTien& x)

Cây

Bảng băm

Đồ thị



Cây

Bảng băm

Đồ thị

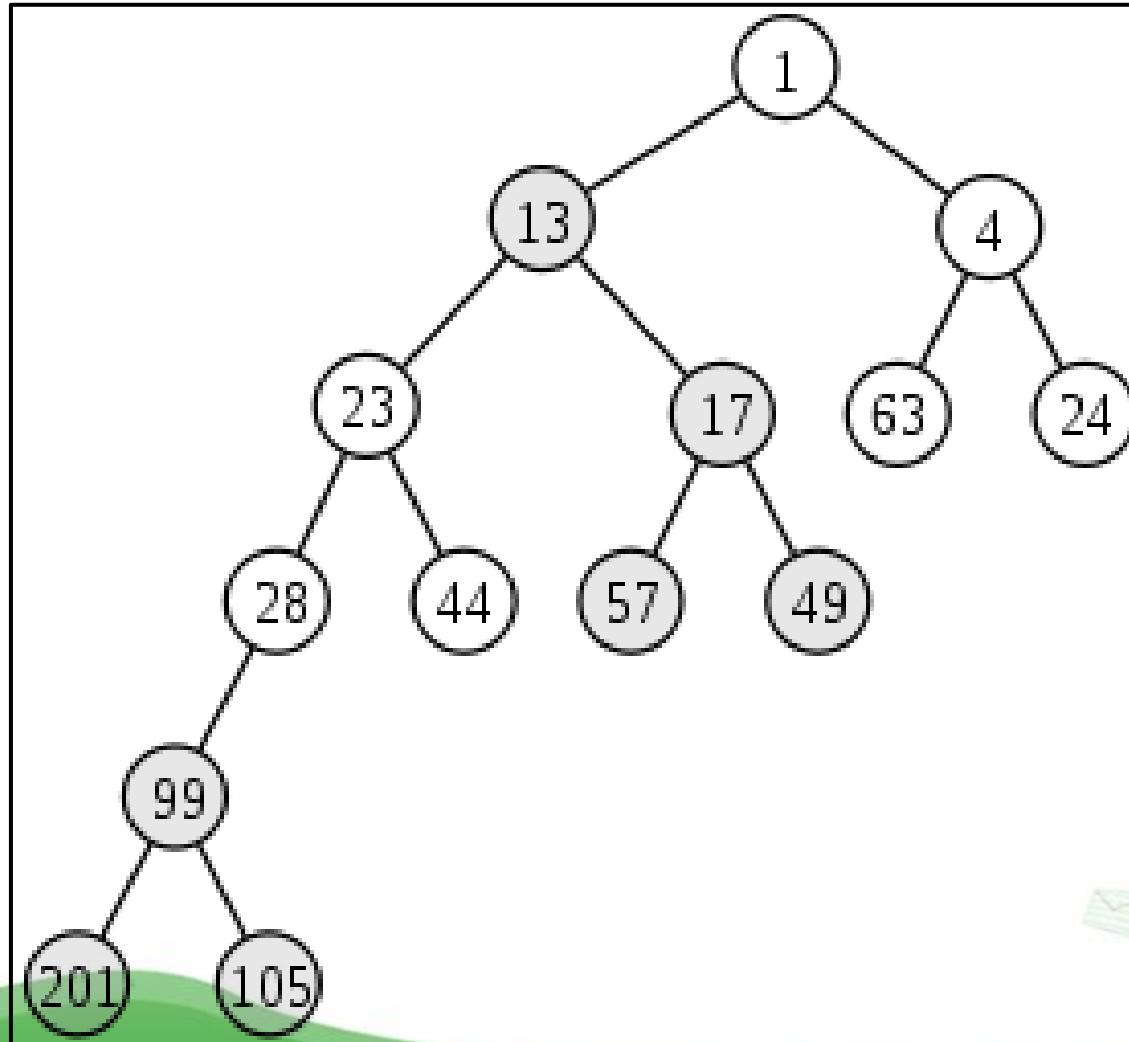
Cây nghiêng (skew heaps)

CÂY NGHIÊNG

Cây

Bảng băm

Đồ thị



CÂY NGHIÊNG

1. Khái niệm:

- Cây nghiêng là cây nhị phân thoả mãn tính chất khoá của dữ liệu trong mỗi đỉnh không lớn hơn khoá của dữ liệu trong các đỉnh con của nó.
- Chúng ta thấy rằng, cây nghiêng có nút gốc là nút có khóa nhỏ nhất tương ứng với phần tử có độ ưu tiên cao nhất trong Hàng ưu tiên. Do đó ta có thể sử dụng cây nghiêng để cài đặt Hàng ưu tiên và xây dựng các phép toán trên cây nghiêng sao cho có thể sử dụng được cho các phép toán trên Hàng ưu tiên.

Cây

Bảng băm

Đồ thị

CÂY NGHIÊNG

Cây

Bảng băm

Đồ thị

2. Các phép toán trên cây nghiêng:

- Phép toán hợp nhất hai cây nghiêng
- Phép toán chèn một phần tử vào Cây nghiêng
- Phép toán loại bỏ nút có khóa nhỏ nhất trong Cây nghiêng

CÂY NGHIÊNG

Cây

Bảng băm

Đồ thị

a. Phép hợp nh \wedge t hai cây nghiêng:

Phép toán trả về cây nghiêng S là hợp nh \wedge t của hai cây nghiêng S1 và S2; nghĩa là chúng ta tạo ra cây nghiêng S có các nút là hợp của các nút trong cây nghiêng S1 và S2.

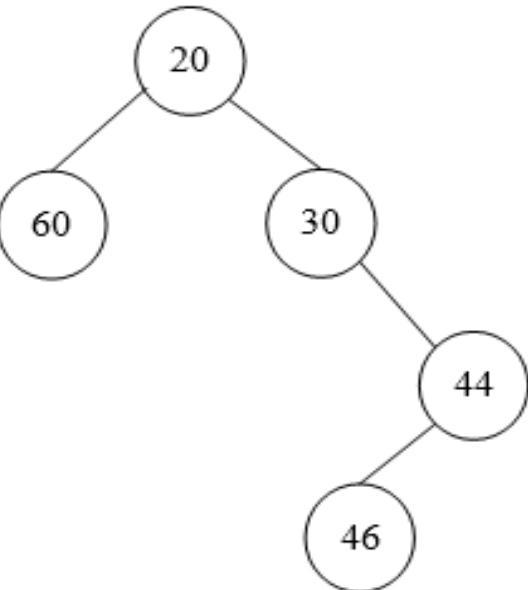
CÂY NGHIÊNG

Cây

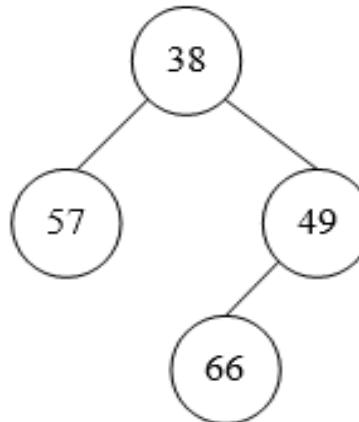
Bảng băm

Đồ thị

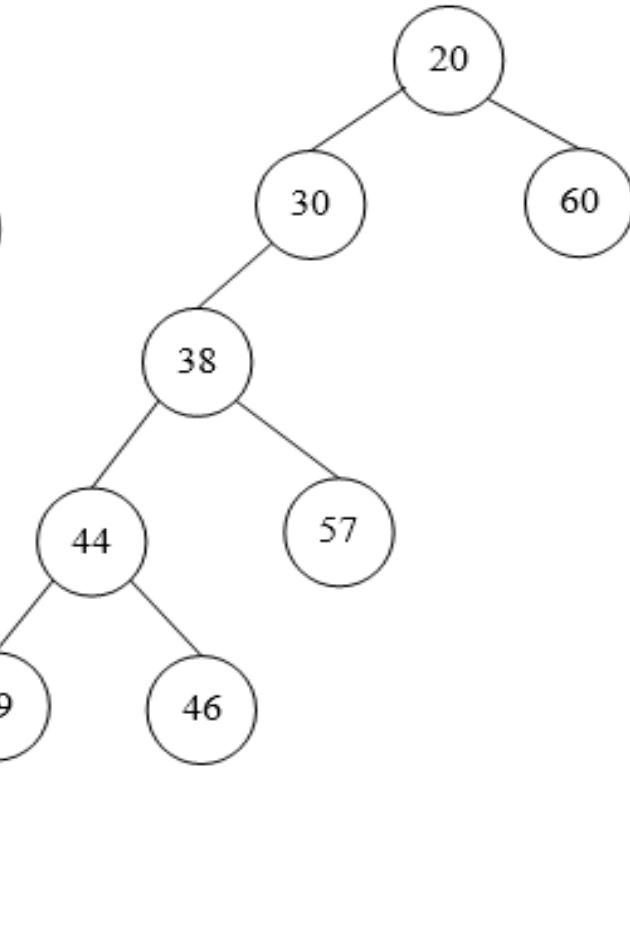
Cây nghiêng S1



Cây nghiêng S2



Cây nghiêng S



Giải thuật

HopNhat(S, S1, S2)

Bước 1: Nếu khóa của nút gốc trong cây nghiêng S1 lớn hơn khóa của nút gốc trong cây nghiêng S2 thì thực hiện hoán đổi hai cây nghiêng S1 và S2

Bước 2:

Nút gốc S là nút gốc của S1 và S.Left=S1.Left (nhánh trái của S là nhánh trái của S1).

HopNhat(S.Right, S2, S1.Right).

Bước 3: Hoán đổi hai cây nghiêng S.Left với S.Right.

Cây

Bảng băm

Đồ thị

CÂY NGHIÊNG

Cây

Bảng băm

Đồ thị

b. Phép chèn một khóa mới vào cây nghiêng:

Phép toán chèn phần tử a vào cây nghiêng S là phép toán hợp nhất giữa Cây nghiêng S và cây nghiêng có một nút chứa phần tử dữ liệu a.

c. Phép toán loại bỏ node có khóa nhỏ nhất trong cây nghiêng:

Nút gốc trong cây nghiêng là nút có khóa nhỏ nhất nên Phép toán loại bỏ khóa nhỏ nhất trong cây nghiêng S tương ứng với việc xóa nút gốc trong cây nghiêng nên phép toán loại bỏ nút có khóa nhỏ nhất là phép toán hợp nhất giữa hai cây con trái và cây con phải của cây nghiêng S.

CÂY NGHIÊNG

c. Phép toán loại bỏ node có khóa nhỏ nhất trong cây nghiêng:

Nút gốc trong cây nghiêng là nút có khóa nhỏ nhất nên Phép toán loại bỏ khóa nhỏ nhất trong cây nghiêng S tương ứng với việc xóa nút gốc trong cây nghiêng nên phép toán loại bỏ nút có khóa nhỏ nhất là phép toán hợp nhất giữa hai cây con trái và cây con phải của cây nghiêng S.

Cây

Bảng băm

Đồ thị

CÂY NGHIÊNG

3. Cấu trúc:

```
typedef int TYPEINFO;
struct NODE
{
    TYPEINFO data;
    NODE* left;
    NODE* right;
};
typedef NODE* NODEPTR;
```

Cấu trúc dữ liệu cho Hàng ưu tiên: chúng ta khai báo một biến có kiểu NODEPTR tương ứng là một Hàng ưu tiên

Cây

Bảng băm

Đồ thị

CÂY NGHIÊNG

4. Các thao tác:

+ Phép toán kiểm tra Hàng ưu tiên rỗng:

int kiemtraRong(NODEPTR r)

+ Phép toán chèn một phần tử vào Hàng ưu tiên:

void chen(NODEPTR& s, TYPEINFO a)

+ Phép toán tham chiếu đến phần tử đầu tiên trong Hàng ưu tiên:

void getPhantuDautien(NODEPTR r, TYPEINFO& a)

+ Phép toán loại bỏ một phần tử khỏi Hàng ưu tiên:

void xoa(NODEPTR& r)

Cây

Bảng băm

Đồ thị



Cây

Bảng băm

Đồ thị

B-Cây

Lan man

- Tốc độ đọc ghi RAM: 6Gb->17Gb/s (DDR3)
- Tốc độ đọc ghi ổ cứng:
 - HDD 7200 RPM: 60Mb – 80Mb/s
 - SSD: 200Mb/s (max 550Mb/s)
 - USB 3.0: lý thuyết 300Mb/s, thực tế 50Mb/s

Dễ dàng nhận thấy rằng sự thống trị về tốc độ đọc và ghi của Ram là không thể chối cãi. Ram dung lượng lớn nhất hiện giờ khoảng 16GB. CPU mạnh nhất có khoảng 12 slot RAM.

Như vậy một máy tính mạnh nhất hiện nay có thể xử lý 1 dữ liệu lên đến 192 GB. (**Enough for us now?**)

Cây

Bảng băm

Đồ thị

B-CÂY

Cây

Bảng băm

Đồ thị

Trong hầu hết các cây tìm kiếm tự cân bằng như AVL và cây Đỏ đen, dữ liệu xử lý được lưu trong bộ nhớ trong (RAM). Tuy nhiên trong trường hợp số lượng dữ liệu khổng lồ mà không thể load vào bộ nhớ trong, khi đó dữ liệu được đọc từ trực tiếp đĩa dưới dạng các khối (block). Thời gian truy xuất đĩa rất cao so với thời gian truy xuất bộ nhớ chính. Ý tưởng chính của việc sử dụng B-Trees là giảm số lần truy cập đĩa. Các thao tác trên B-Trees (tìm kiếm, chèn, xoá...) yêu cầu truy cập đĩa $O(h)$ trong đó h là chiều cao của cây.



❑ ĐỊNH NGHĨA

B-CÂY

Cây

Bảng băm

Đồ thị

Không như cây nhị phân, mỗi node của B-Cây có thể chứa nhiều key và có thể có nhiều hơn 2 cây con. B-cây được phát biểu bởi Bayer and McCreight(1972) với tên gọi là *m-way Search Tree*. Sau này được gọi là B-Tree (B-cây).

1. Định nghĩa:

B-Cây là một cây tìm kiếm tự cân bằng mà mỗi node có thể có nhiều khóa và hơn 2 cây con.

Số khóa và con của mỗi node phụ thuộc vào bậc của cây. Mọi B-cây đều có bậc kèm.

B-CÂY

Cây

Bảng băm

Đồ thị

B-cây bắc m phải thỏa các tính chất sau:

- Tất cả các node lá phải có cùng độ sâu
- Tất cả các node trừ node gốc phải có ít nhất $\left(\frac{m}{2}\right) - 1$ và tối đa $m - 1$ khóa.
- Tất cả các node không phải lá và gốc phải có ít nhất $m/2$ cây con.
- Nếu node gốc không là node lá thì nó phải có ít nhất 2 con.
- Nếu 1 node không là lá có $n - 1$ khóa thì có n cây con
- Tất cả khóa trong cùng node phải sắp tăng dần

B-CÂY

Cấu trúc của B-Cây:

Khai báo:

```
#define ORDER 5

typedef struct tagNODE
{
    int      NumTree;      // số cây con của node hiện hành
    int      Key[ORDER-1];  // mảng lưu trữ các khoá của node
    tagNODE* Branch[ORDER]; // các con trỏ chỉ đến các node con
} NODE, *NODEPTR, *BTREE; // con trỏ node

NODEPTR Root ;// con trỏ node gốc
```

Cây

Bảng băm

Đồ thị

B-CÂY

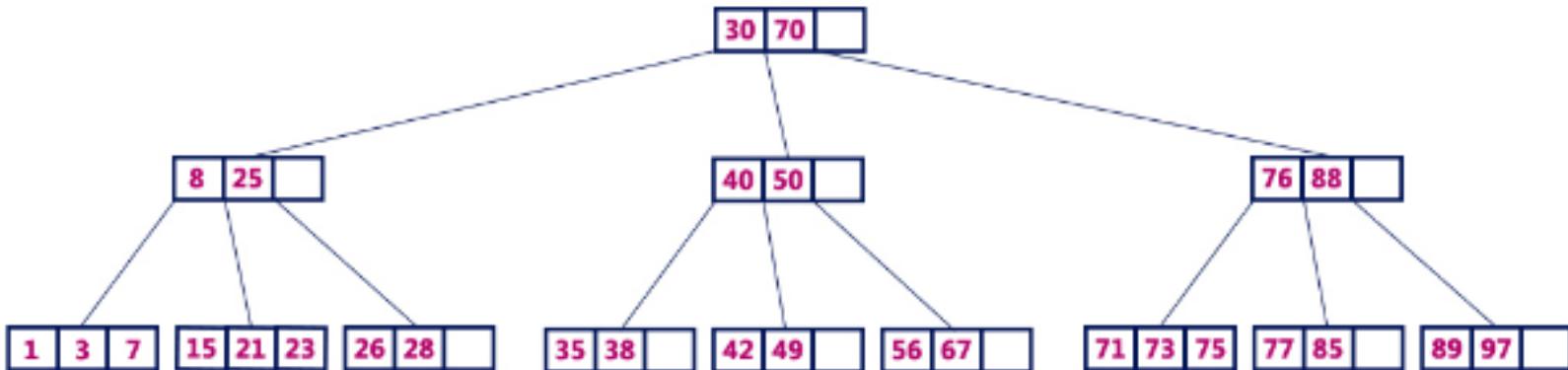
Ví dụ B-cây bậc 4 thì mỗi node có tối đa 3 khóa và tối đa 4 cây con.

Cây

Bảng băm

Đồ thị

B-cây bậc 4



❑ CÁC THAO TÁC TRÊN B-CÂY

B-CÂY

2. Các thao tác trên B-cây:

- Tìm kiếm
- Thêm
- Xóa

Cây

Bảng băm

Đồ thị

❑ TÌM KIẾM TRÊN B CÂY

Tìm kiếm trên phần tử s trên B cây T

- **B1:** Nếu $T==NULL \rightarrow$ không tìm thấy ngược lại qua bước 2.
- **B2:** So sánh s và khóa đầu tiên của node gốc
- **B3:** Nếu giá trị bằng nhau \rightarrow tìm thấy ngược lại qua bước 4.
- **B4:** nếu s nhỏ hơn khóa \rightarrow tìm bên con trái
- **B5:** nếu s lớn hơn khóa so sánh s với khóa kế tiếp trong node, lặp lại bước 3 cho đến khi tìm thấy hoặc s lớn hơn khóa cuối cùng trong node thì tìm bên cây con phải.

Cây

Bảng băm

Đồ thị

❑ THÊM PHẦN TỬ VÀO B CÂY

Thêm phần tử s vào B cây

- **B1:** Nếu cây rỗng, tạo node mới với khóa là s và cho nó làm node gốc, ngược lại qua bước 2.
- **B2:** Tìm node sẽ thêm khóa s vào theo logic của cây nhị phân tìm kiếm.
- **B3:** Nếu node lá đó còn chỗ trống thì thêm s vào chỗ trống đó theo đúng thứ tự tăng dần.
- **B4:** Nếu node cần thêm khóa vào đã đầy:
 - Thêm s vào dây khóa, dựa vào khóa **x** ở vị trí giữa chia dây đó làm 2 dây không chứa **x**. Chia 2 dây đó cho 2 node (node cũ + 1 node mới) và khóa **x** được thêm vào node cha của node đang xét. Nếu node cha là NULL thì x là gốc mới.

Bảng băm

Đô thị

Cây

Nén tập tin

Thêm phần tử s vào B cây

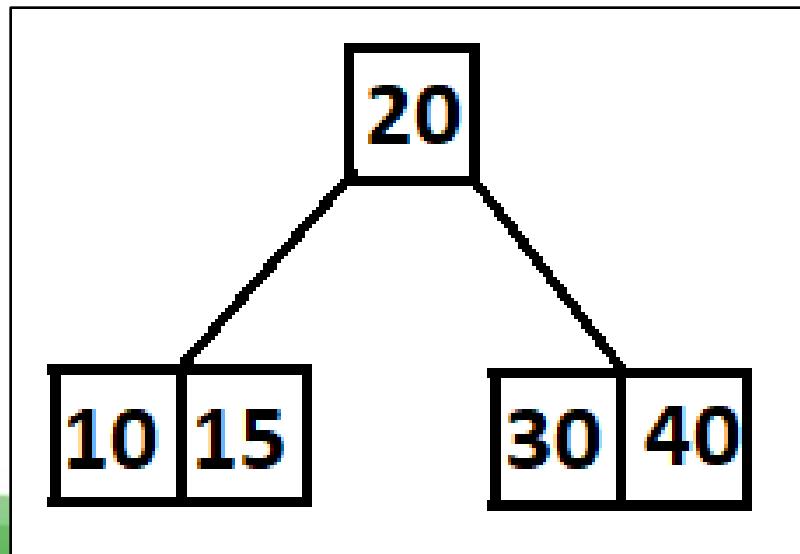
Ví dụ: Xem quá trình tạo **B-Tree bậc 5** từ dãy các khóa sau : 20, 40, 10, 30, 15, 35, 7, 26, 18, 22, 5, 42, 13, 46, 27, 8, 32, 38, 24, 45, 25.

sau khi thêm
20,40,10,30

10	20	30	40
----	----	----	----

Thêm phần tử s vào B cây

Khi thêm vào 15 thì node này bị đẩy, do đó trường hợp này tạo thành 2 node mới : phần tử ở giữa là 20 bị đẩy lên tạo thành một node mới, các phần tử còn lại chia cho 2 node : node cũ chứa 10, 15 và node mới thứ 2 chứa 30, 40



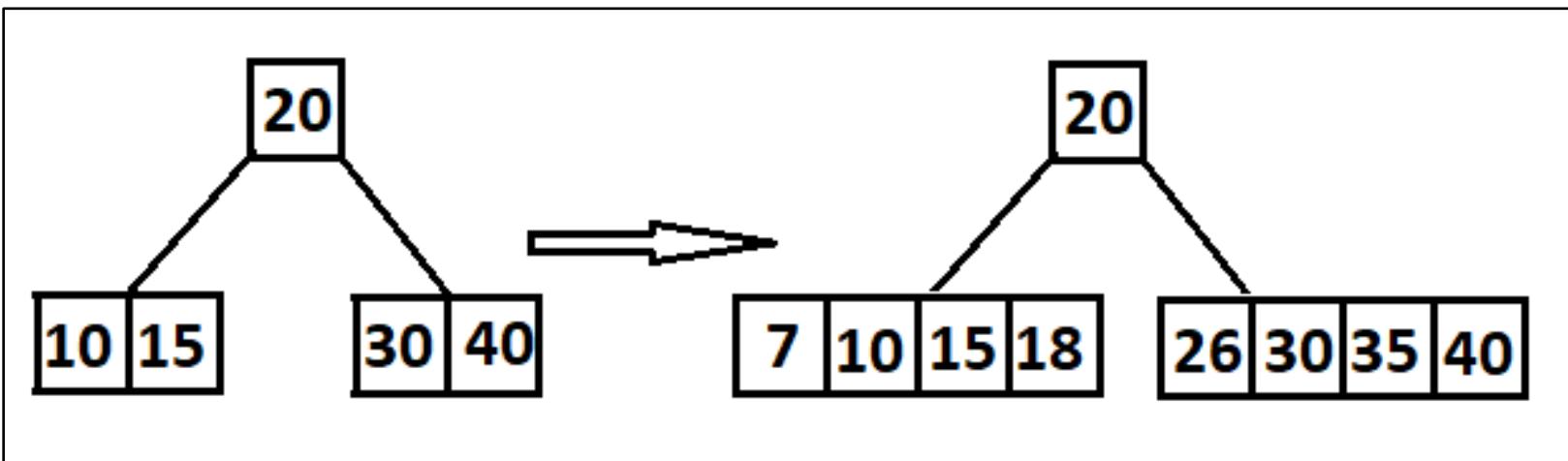
Thêm phần tử s vào B cây

Cây

Bảng băm

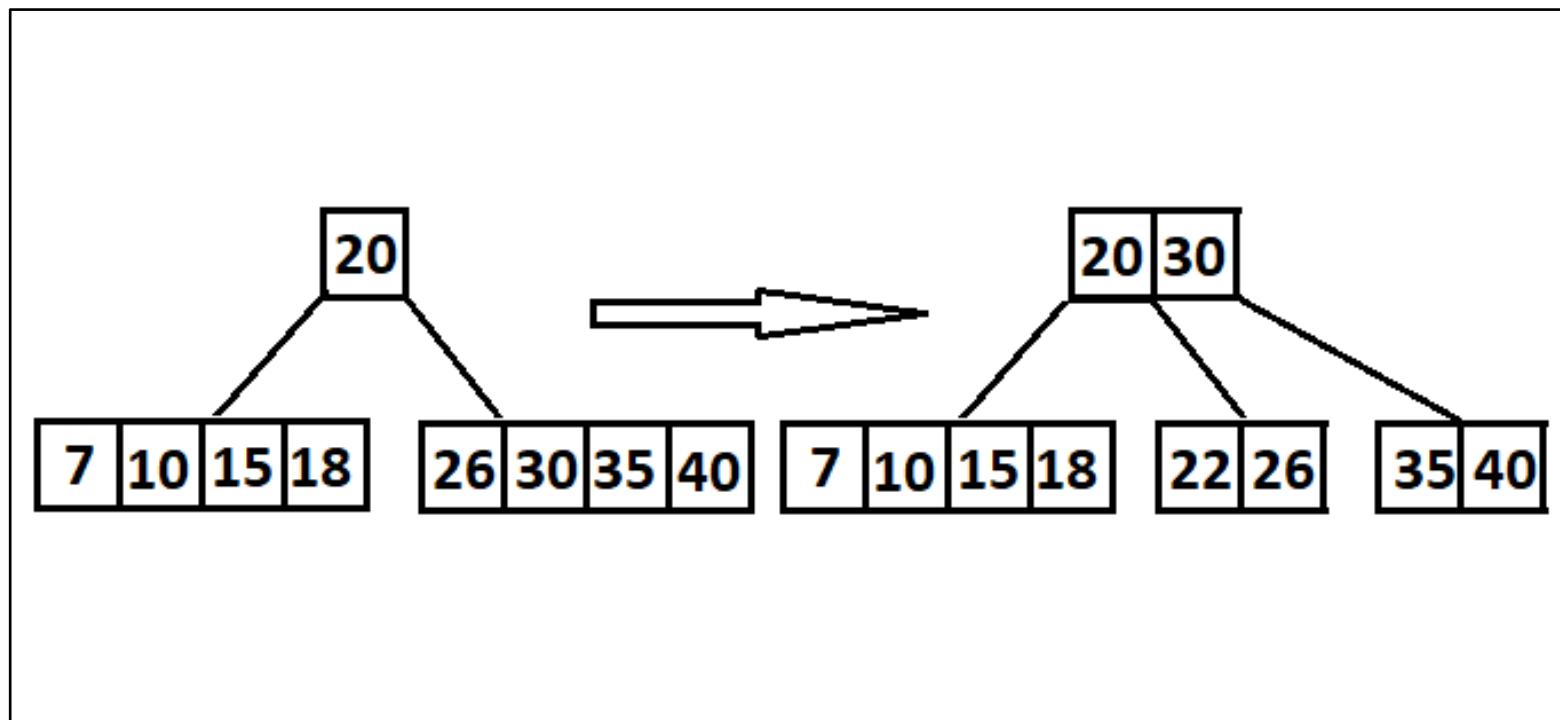
Đồ thị

Thêm vào các khóa 35, 7, 26 và 18 chỉ đơn giản là việc bổ sung các khóa vào các node có sẵn:



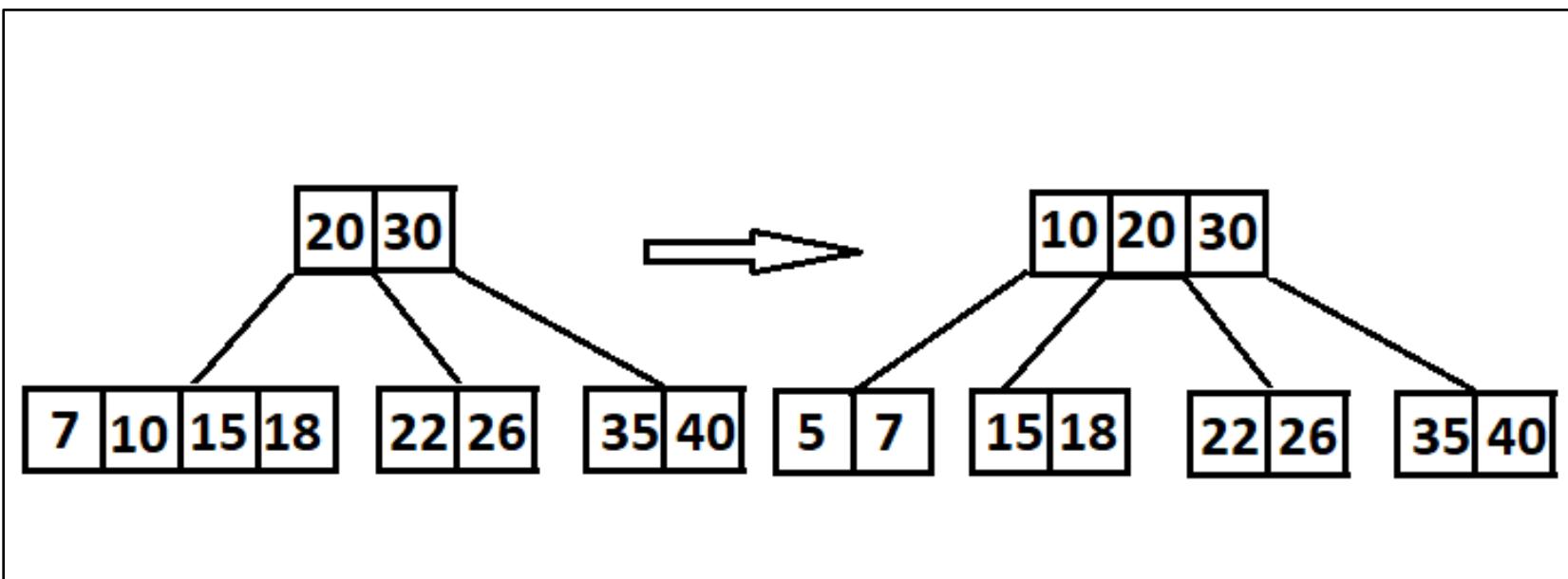
Thêm phần tử s vào B cây

Khi thêm khóa 22 cũng có sự đâm node dẫn đến việc tách node, phần tử ở giữa là 30 sẽ bị thêm vào node cha chứa khóa 20.



Thêm phần tử s vào B cây

Thêm vào 5 cũng có sự đâm node (node đang chứa 4 khóa 7, 10, 15, 18) dẫn đến việc tách node, khóa giữa 10 sẽ được thêm vào node cha đang có dãy khóa 20,30:



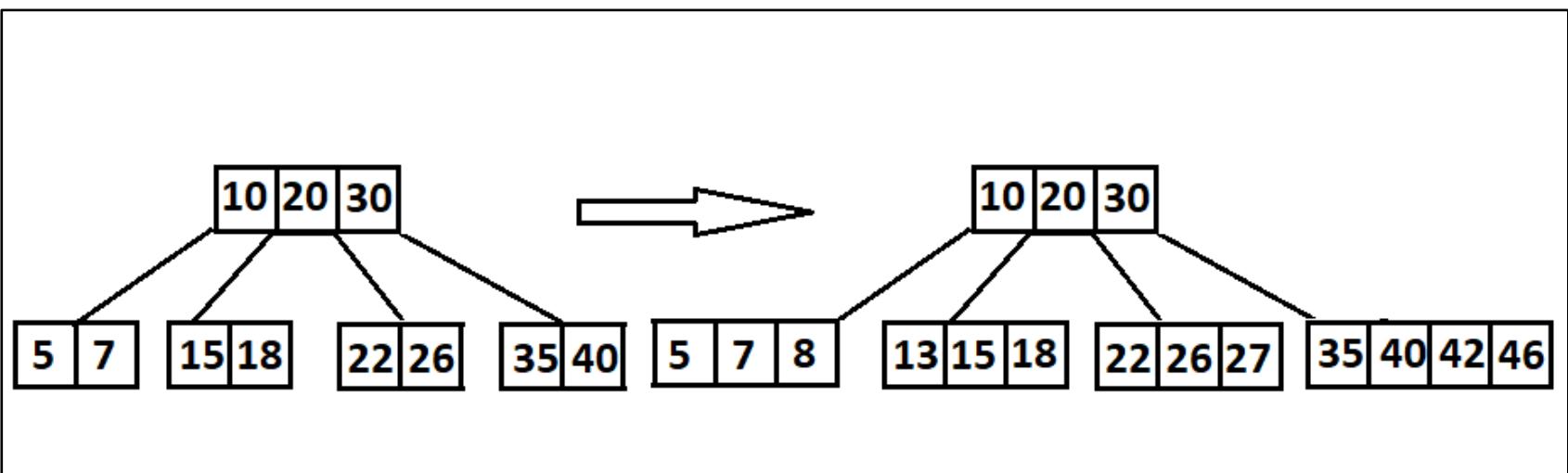
Cây

Bảng băm

Đồ thị

Thêm phần tử s vào B cây

Thêm vào các khóa 42, 13, 46, 27 và 8 chỉ đơn giản là quá trình bổ dung khóa vào các node có sẵn:



Cây

Bảng băm

Đồ thị

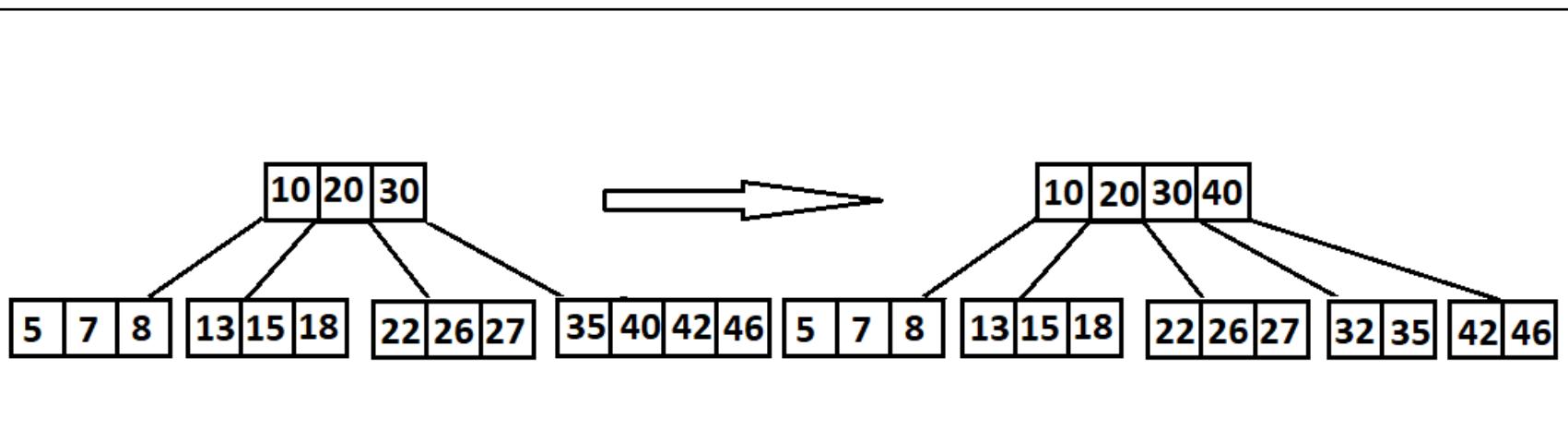
Thêm phần tử s vào B cây

Khi thêm 32 có sự tách node phần tử giữa 40 được thêm vào node cha đang chứa dãy khóa 10,20,30

Cây

Bảng băm

Đồ thị



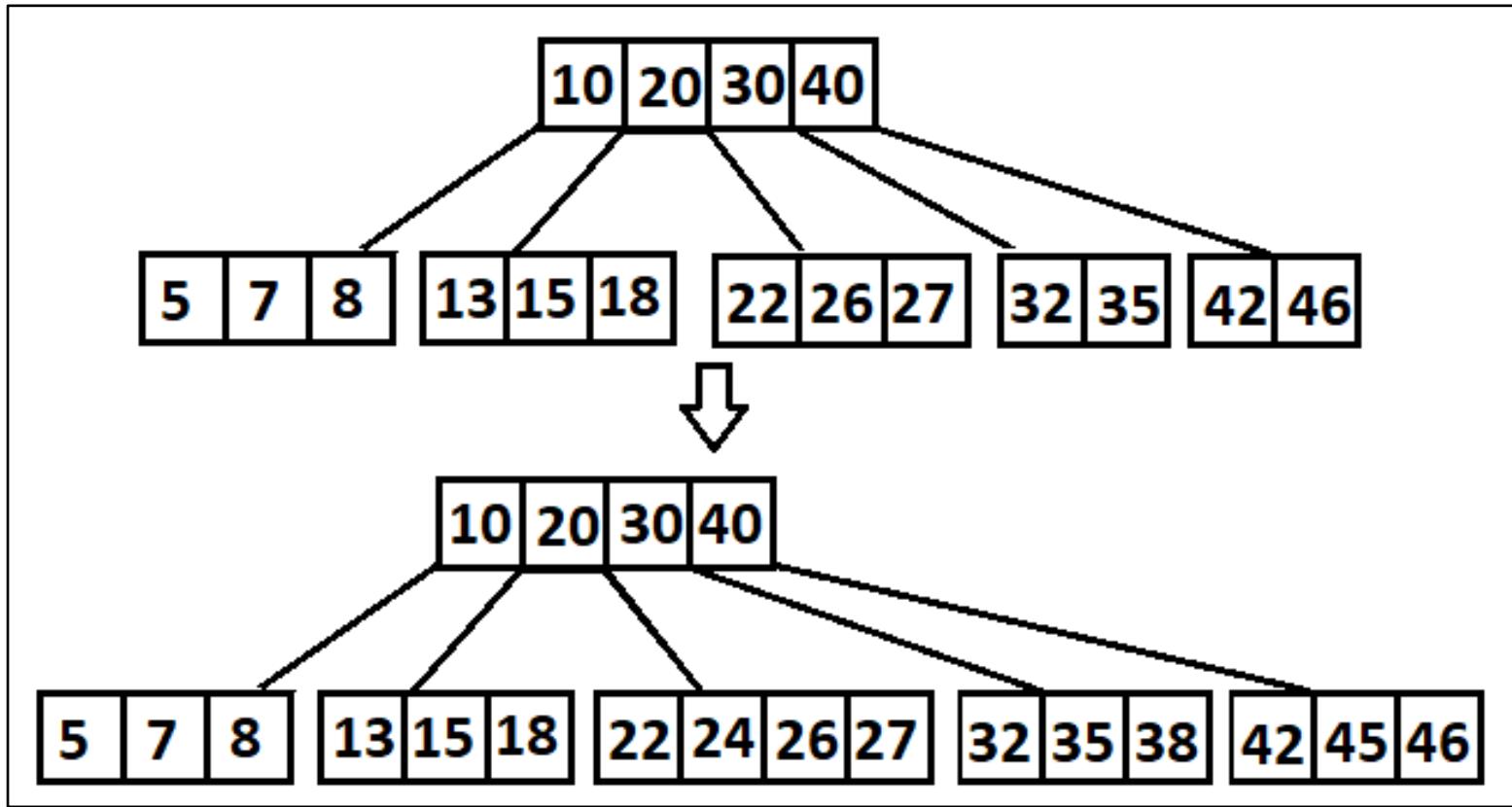
Thêm phần tử s vào B cây

Cây

Bảng băm

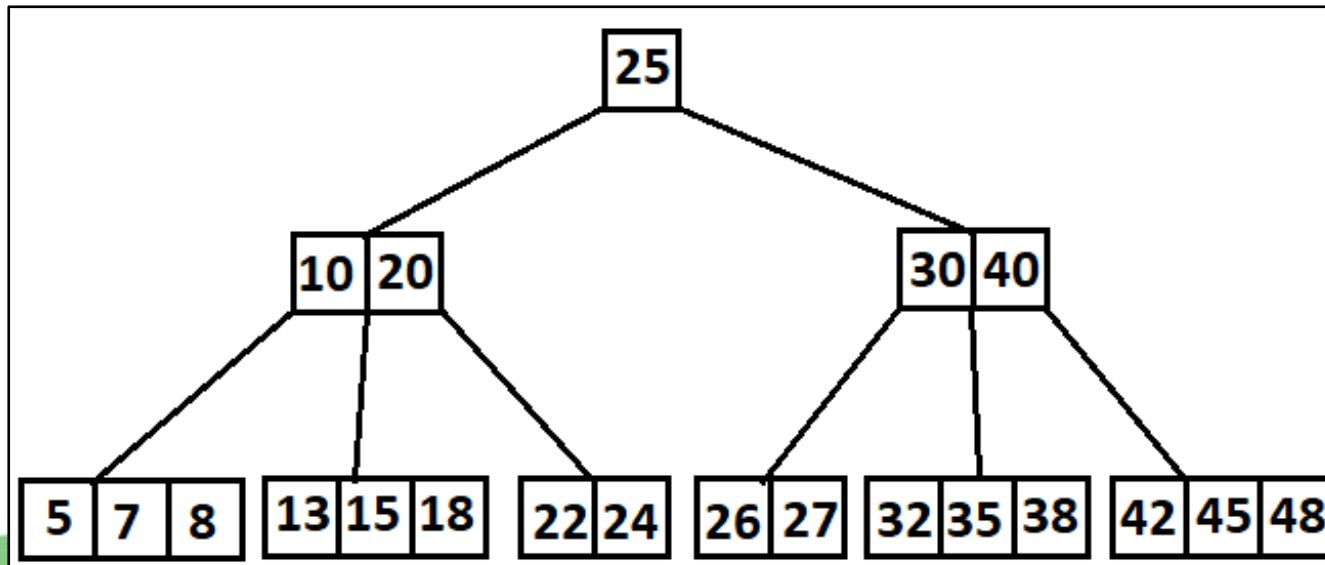
Đồ thị

Thêm vào 38, 24 và 45 chỉ đơn giản là quá trình bổ dung khóa vào các node có sẵn.



Thêm phần tử s vào B cây

Thêm 25 vào có sự tách node, node giữa 25 được thêm vào node cha đang chứa dãy khóa 10,20,30,40 lại dẫn đến sự tách node, node giữa 25 lại được thêm vào node cha lúc này là NULL nên trở thành node gốc mới:



Thêm phần tử s vào B cây

Thêm 25 vào có sự tách node, node giữa 25 được thêm vào node cha đang chứa dãy khóa 10,20,30,40 lại dẫn đến sự tách node, node giữa 25 lại được thêm vào node cha lúc này là NULL nên trở thành node gốc mới:

Cây

Bảng băm

Đồ thị

XÓA PHẦN TỬ KHỎI BÃY

B-CÂY

Cây

Bảng băm

Đồ thị

Xóa một node khỏi B-Cây

1. Nếu khóa cần xóa nằm ở nút lá, và việc xóa khóa này không làm cho nút lá có ít hơn số khóa tối thiểu ($m/2$ khóa), chúng ta có thể xóa ngay khóa này.
2. Nếu khóa cần xóa không nằm ở nút lá, theo tính chất của B-tree, khóa có giá trị gần nhất của khóa này (khóa thay thế) phải nằm ở nút lá. Trong trường hợp này, chúng ta đưa khóa có giá trị gần nhất từ nút lá lên thay thế cho khóa đã xóa, xóa khóa thay thế.

B-CÂY Can xem lai khi cha thieu khoa

Trong trường hợp nút lá còn lại **quá ít** khóa thì, xét (các) nút **anh em kế cận** nút đang xét:

3. Nếu một trong các nút anh em kế cận có số lượng khóa nhiều hơn số lượng tối thiểu, đưa một khóa của nút anh em lên nút cha và đưa một khóa ở nút cha xuống nút đang xét

4. Nếu tất cả các nút anh em kế cận đều có số lượng khóa là tối thiểu, chọn một nút anh em kế cận và hợp nhất nút anh em này với nút đang xét (và khóa tương ứng ở nút cha). Nếu nút cha trở nên thiếu khóa, lặp lại quá trình này.

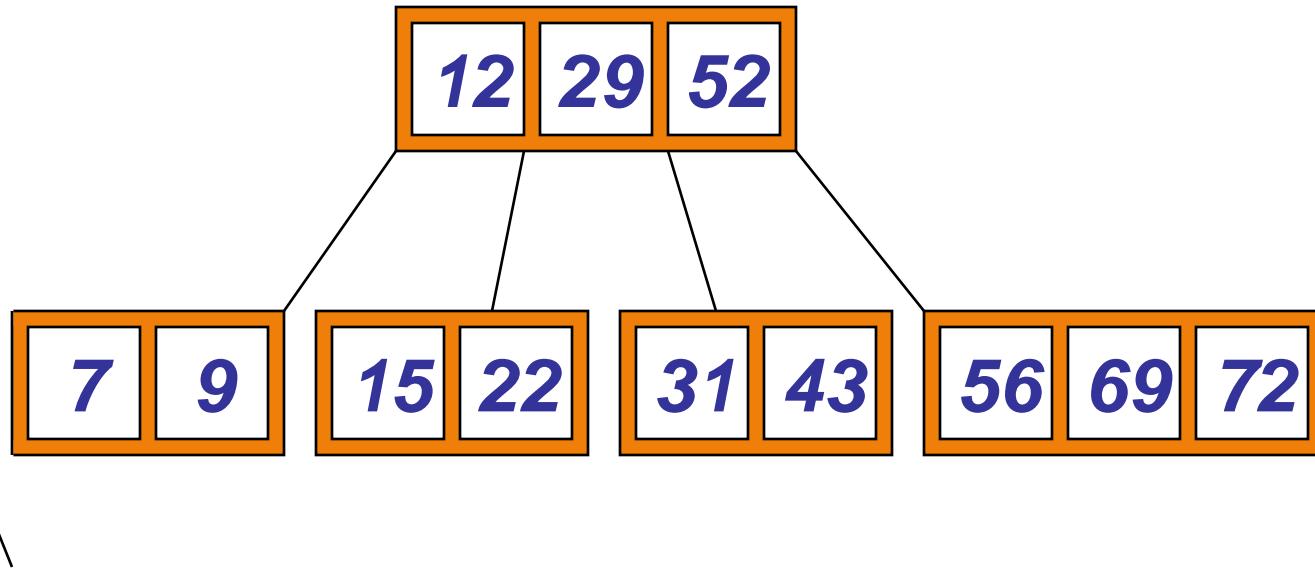
Cây

Bảng băm

Đồ thị

Trường hợp #1: Xóa khóa ở nút lá

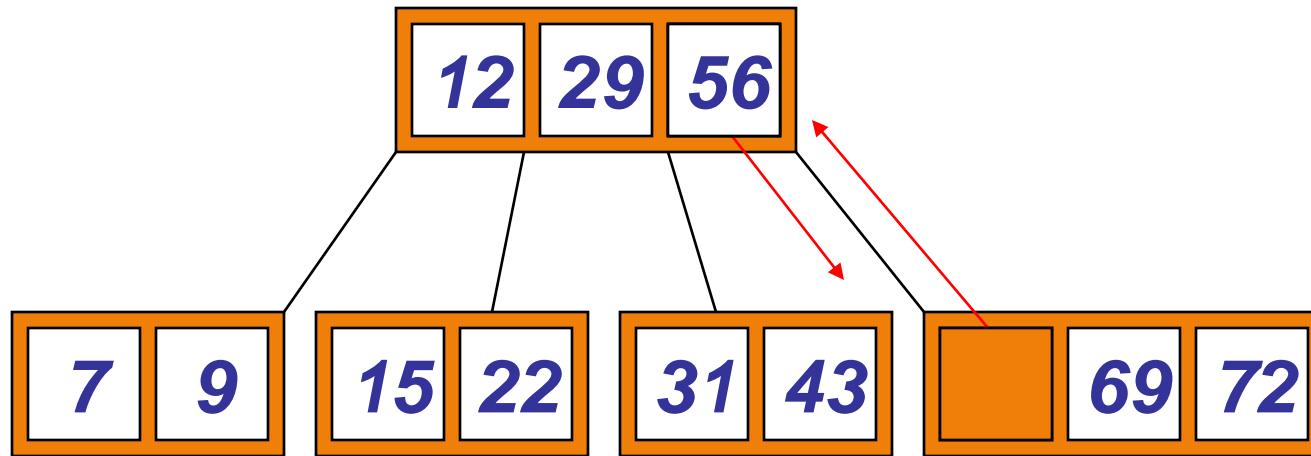
Xóa B-Tree bậc 5 đã xây dựng sẵn..



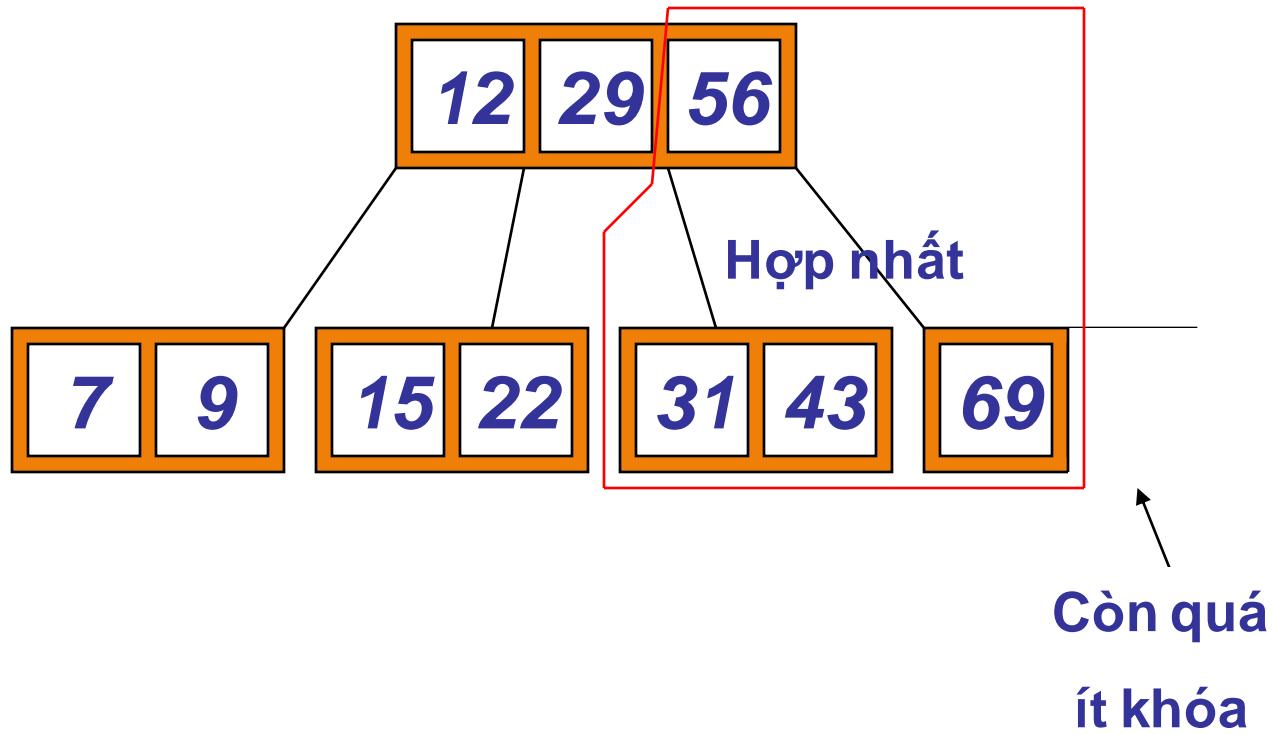
Xóa 2: Nút lá vẫn còn đủ số lượng
phần tử theo yêu cầu



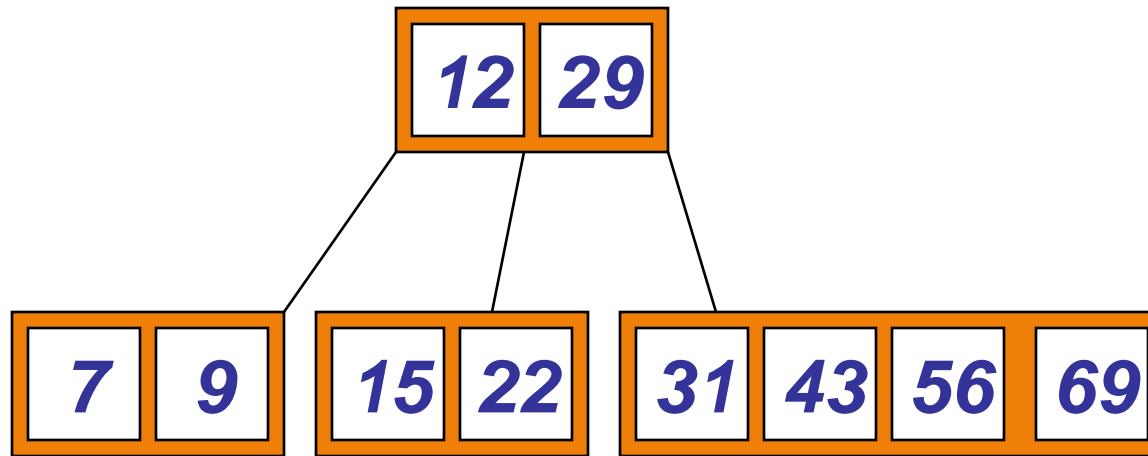
Trường hợp #2: Xóa khóa không ở nút lá



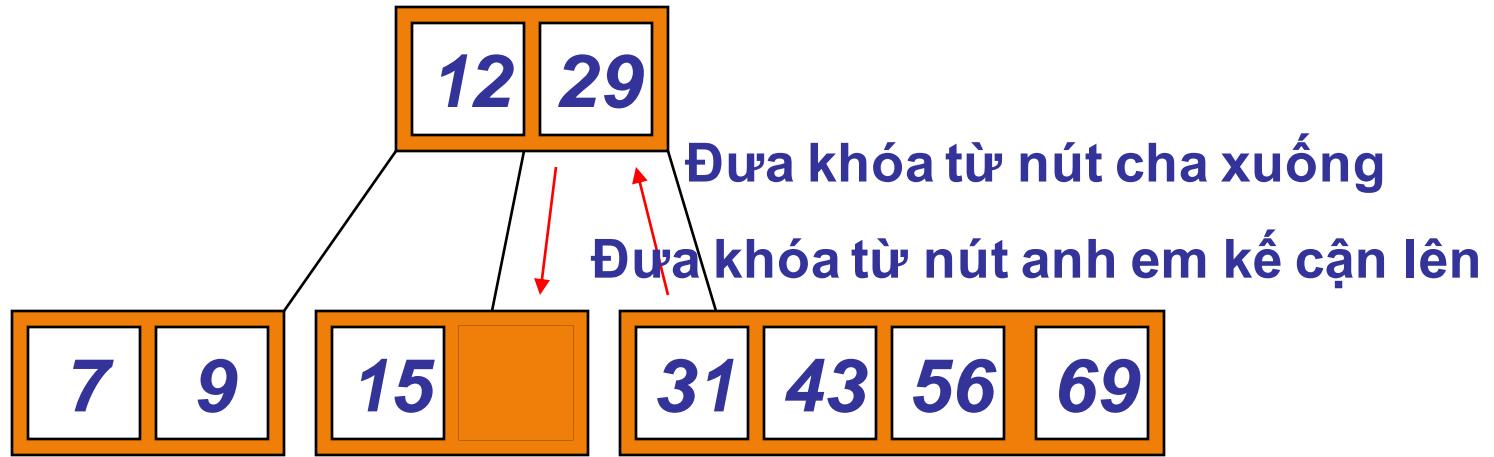
Trường hợp #4: Nút đang xét và nút anh em kế cận đều còn quá ít khóa



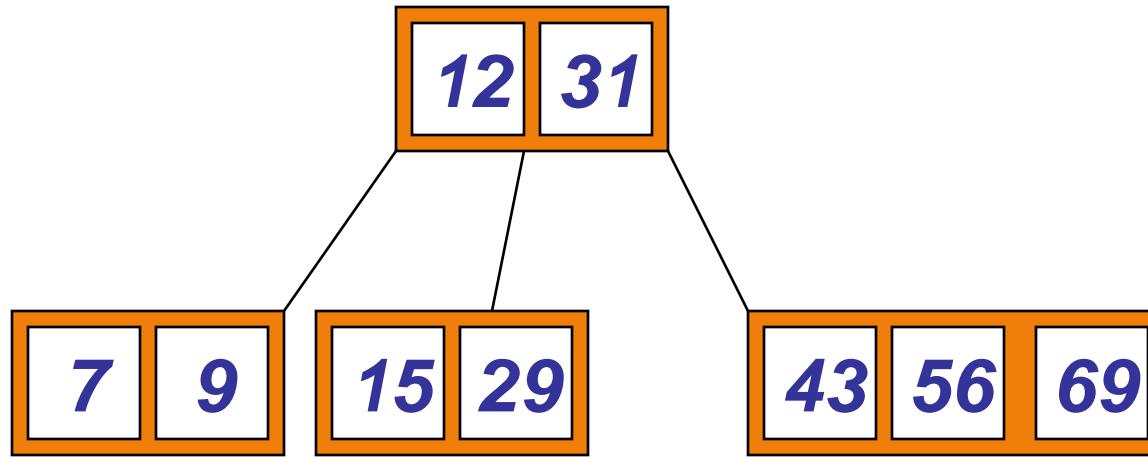
Trường hợp #4: Nút đang xét và nút anh em kế cận đều còn quá ít khóa



Trường hợp #3: Nút anh em kế cận còn đủ khóa để bổ sung



Trường hợp #3: Nút anh em kế cận còn đủ khóa để bổ sung



Bài tập

1. Cho B-tree bậc 5 gồm các khóa sau (chèn vào theo thứ tự): 3, 7, 9, 23, 45, 1, 5, 14, 25, 24, 13, 11, 8, 19, 4, 31, 35, 56
 - Thêm khóa: 2, 6, 12
 - Xóa khóa: 4, 5, 7, 3, 14
2. Khởi tạo B Tree bậc 7 với các thao tác
Insert:34, 12, 55, 21, 6, 84, 5, 33, 15, 74, 54, 28, 10, 19
 - Sau đó thực hiện chuỗi thao tác:
Insert(11), Delete(15), Delete(6),
Insert(98), Delete(34), Delete(5)





Cây

Bảng băm

Đồ thị

BẢNG BĂM

Bảng băm

- Ôn tập Danh sách liên kết đơn
- Các khái niệm và ý nghĩa của Bảng băm
- Hàm băm
- Bảng băm đóng
- Bảng băm mở

Cây

Bảng băm

Đồ thị

ÔN TẬP DANH SÁCH LIÊN KẾT ĐƠN

Danh sách liên kết đơn

- Danh sách liên kết là danh sách mà các phần tử được kết nối với nhau thông qua một vùng gọi là vùng liên kết theo trình tự tuyến tính.
- Mỗi nút (phần tử) trong danh sách liên kết đơn có hai vùng. Vùng thứ nhất chứa dữ liệu và vùng thứ hai là vùng liên kết dùng để kết nối các nút với nhau.
- Ưu điểm của cấu trúc danh sách liên kết đơn là chúng ta sẽ không bị giới hạn về số lượng phần tử của danh sách khi làm việc với các loại danh sách mà số lượng phần tử là không cố định.

Cây

Bảng băm

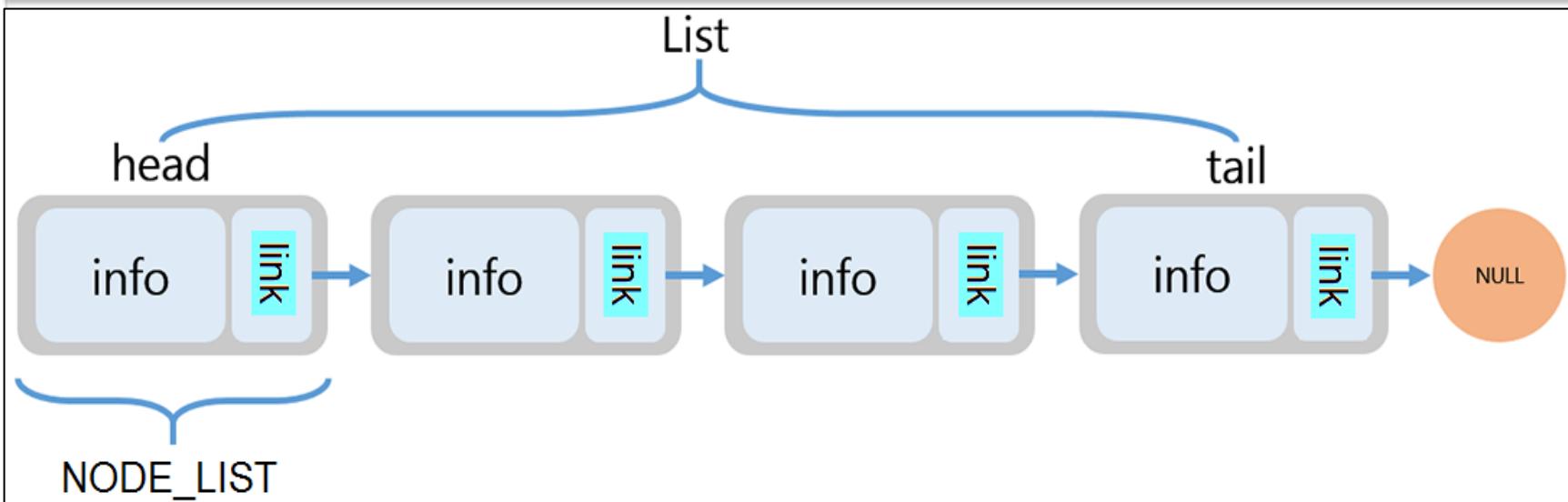
Đồ thị

Danh sách liên kết đơn

Cây

Bảng băm

Đồ thị



- Cấu trúc dữ liệu
- Duyệt qua một danh sách liên kết đơn
- Tìm kiếm, thêm, xóa một phần tử trong danh sách

Ví dụ 1:**Cây****Bảng băm****Đồ thị**

33	95	72	47
60	4	46	13
49	37	71	2
15	22	14	32
27	17	8	16
9	19	91	73
81	6	76	11

- Lưu trữ các số nguyên trên bảng danh sách đặc và danh sách liên kết đơn. Hãy so sánh ưu khuyết điểm khi tìm kiếm, xóa, lưu trữ?



CÁC KHÁI NIỆM VÀ Ý NGHĨA CỦA BẢNG BĂM

bảng băm

Cây

Bảng băm

Đồ thị

- Khi cần tìm kiếm, chèn, xóa 1 phần tử trên danh sách, chúng ta mất khá nhiều thời gian hoặc bị hạn chế về kích thước danh sách.

⇒ **Có cách nào có thể cải tiến được vấn đề này?**

- Dựa trên ý tưởng: chọn một thuộc tính nào đó của đối tượng dữ liệu làm khóa, chúng ta biến đổi giá trị khóa thành một số và sử dụng số này để làm chỉ mục cho đối tượng trong một danh sách đặc.

Khái niệm bảng băm

- Trong khoa học máy tính, **bảng băm** là một cấu trúc dữ liệu sử dụng hàm băm để ánh xạ từ giá trị xác định, được gọi là khóa (ví dụ như tên của một người) đến giá trị tương ứng dùng để xác định vị trí lưu trữ thích hợp cho đối tượng chứa khóa đó trong một danh sách đặc(mảng). Do đó, bảng băm là một mảng kết hợp. Hàm băm được sử dụng để chuyển đổi từ khóa thành chỉ số (giá trị băm) trong mảng lưu trữ các giá trị tìm kiếm.

Cây

Bảng băm

Đồ thị

❑ HÀM BĂM

Hàm băm

- **Hàm băm (Hash function):**

Là giải thuật nhằm sinh ra các giá trị băm tương ứng với mỗi khối dữ liệu (có thể là một số, một chuỗi kí tự, một đối tượng trong lập trình hướng đối tượng, v.v...). Giá trị trả về của hàm băm đóng vai gần như một **khóa** để phân biệt các khối dữ liệu, tuy nhiên, người ta chấp nhận tượng trùng khóa hay còn gọi là sự đụng độ và cố gắng cải thiện giải thuật để giảm thiểu sự đụng độ đó.

Cây

Bảng băm

Đồ thị

Hàm băm

- **Hàm băm một số nguyên:**

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

```
// hàm băm
int funcHash(TYPEINFO a,int size)
{
    return a%size;
}
```

Cây

Bảng băm

Đồ thị

Hàm băm

- **Hàm băm cho giá trị khóa là chuỗi:**

- Để băm cho các xâu ký tự ta cần biến đổi các xâu thành các số nguyên.
- Các ký tự trong bảng mã ASCIIII gồm 128 ký tự được đánh số từ 0 đến 127, do 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 hệ đếm cơ số 10, chúng ta sẽ chuyển được một xâu ký tự thành 1 số nguyên.

Cây

Bảng băm

Đồ thị

Hàm băm

- Ví dụ:

“NOTE” được chuyển thành 1 số nguyên:

$$\begin{aligned} \text{NOTE} &\rightarrow N * 128^3 + O * 128^2 + T * 128^1 + E * 128^0 \\ &= 78 * 128^3 + 79 * 128^2 + 84 * 128^1 + 69 * 128^0 \end{aligned}$$

Tuy nhiên với những xâu ký tự tương đối dài thì kết quả lại là một số nguyên vô cùng lớn, đôi khi vượt quá khả năng lưu trữ của máy tính.

Cây

Bảng băm

Đồ thị

Hàm băm

Trong thực tế một xâu ký tự thông thường được lập nên từ 26 chữ cái, 10 chữ số và một vài ký tự đặc biệt, do đó ta có thể thay 128 bằng 37 để tính số nguyên ứng với xâu ký tự.

$$\begin{aligned} \text{NOTE} &\rightarrow N * 37^3 + O * 37^2 + T * 37^1 + E * 37^0 \\ &= 78 * 37^3 + 79 * 37^2 + 84 * 37^1 + 69 * 37^0 \end{aligned}$$

Sau khi tính được số nguyên ta sẽ dùng hàm băm một số nguyên để tính ra chỉ mục cho xâu trong bảng băm.

Cây

Bảng băm

Đồ thị

Hàm băm

Hàm băm cho một chuỗi:

```
unsigned int funcHash(char* k, int SIZE)
{
    unsigned int value = 0;
    for (int i = 0; i < strlen(k); i++)
        value = 37 * value + k[i];
    return value % SIZE;
}
```

Cây

Bảng băm

Đồ thị

BẢNG BĂM ĐÓNG

Ví dụ

Cây

Bảng băm

Đồ thị

33	95	72	47
60	4	46	13
49	37	71	2
15	22	14	32
27	17	8	16
9	19	91	73
81	6	76	11

- Với dữ liệu trên, ta có thể xây dựng một bảng băm với hàm băm như thế nào?

Bảng băm

- Ta thấy các dữ liệu đã cho đều khác nhau và bé hơn 100 nên đơn giản nhất là chúng ta xây dựng một danh sách đặc có 100 phần tử và sử dụng các giá trị của nó làm chỉ mục tương ứng.

0	1	2	3	...	97	98	99
0	0	0	0	...	0	0	0

- Phần tử thứ $k = 1$ nếu k có trong danh sách, ngược lại là 0

Bảng băm

- Hoặc ta có thể dùng 1 mảng 50 phần tử và lưu trữ như sau:

0	1	2	3	...	97	98	99
0	0	0	0	...	0	0	0

- Phần tử a nếu chia 50 dư k thì a sẽ được lưu tại vị trí thứ k .
- Đây được gọi là **bảng băm đóng**.

Cây

Bảng băm

Đồ thị

Bảng băm

Cây

Bảng băm

Đồ thị

- Tuy nhiên trong **bảng băm đóng**:
 - Có phải lúc nào các thuộc tính khóa cũng khác nhau? (sự đụng độ)
 - Nếu các khóa đã khác nhau thì điều gì xảy ra nếu ta thêm phần tử mới có giá trị 150 vào dữ liệu trong ví dụ 1?
 - Điều gì xảy ra nếu dữ liệu của chúng ta không phải là những số nguyên?
- Những hạn chế này sẽ được giải quyết bằng cấu trúc **bảng băm mở** mà ta sẽ xem xét sau.

BẢNG BĂM MỞ

Bảng băm mở

- Khi 2 đối tượng dữ liệu thông qua hàm băm lại cho cùng 1 kết quả đây gọi là sự va chạm. Để giải quyết sự va chạm này có 2 phương pháp chính là:
 - Phương pháp định địa chỉ mở (**Bảng băm đóng**: thăm dò tuyến tính, thăm dò bình phương,...)
 - Phương pháp tạo dây chuyền (**Bảng băm mở**: sử dụng danh sách liên kết)

Cây

Bảng băm

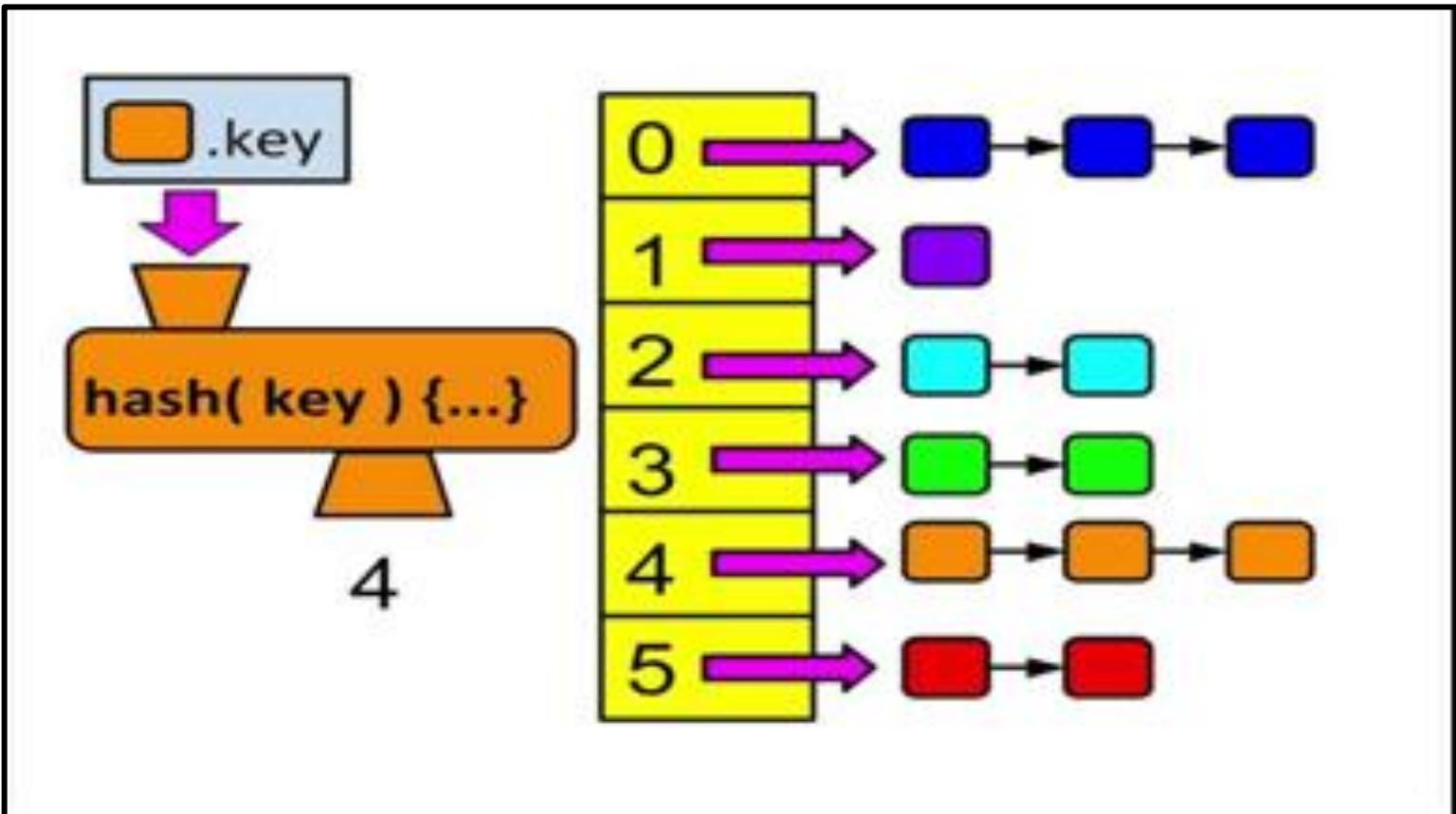
Đồ thị

Bảng băm mở

Cây

Bảng băm

Đồ thị



Bảng băm mở

- Cấu trúc của bảng băm mở:

```
struct OpenHashTable
{
    NODE_LIST* ds[MAX];
    int n;
};
```

Cây

Bảng băm

Đồ thị

Bảng băm mở

- Các phép toán trên bảng băm mở:
 - Tìm kiếm 1 phần tử trong bảng băm
 - Thêm 1 phần tử vào bảng băm
 - Xóa 1 phần tử khỏi bảng băm
- Các thao tác khác:
 - Xuất bảng băm ra màn hình console
 - Xuất bảng băm ra file (text, nhị phân)
 - Đọc bảng băm từ file (text, nhị phân)

Cây

Bảng băm

Đồ thị



Cây

Bảng băm

Đồ thị

ĐỒ THỊ

Các phép toán trên mệnh đề

- Ôn tập xử lý file
- Các khái niệm cơ bản
- Biểu diễn đồ thị (Ma trận kề, danh sách kề)
- Duyệt đồ thị (BFS, DFS)
- Đồ thị liên thông
- Sắp xếp topo
- Đường đi ngắn nhất (Floyd, Dijkstra)
- Cây bao trùm ngắn nhất (Kruskal, Prim)

Cây

Bảng băm

Đồ thị



Cây

Bảng băm

Đồ thị

Ôn tập xử lý file

Ôn tập xử lý file

1. Khai báo và tạo kết nối:

a. Khai báo:

```
FILE *f; //khai bao mot ket noi ten la f
```

b. Tạo kết nối:

`fopen_s(&f,<tên file>, "<ký hiệu mở file để làm gì>");`

`<ký hiệu mở file để làm gì>` xem trong bảng sau

Cây

Bảng băm

Đồ thị

Ôn tập xử lý file

Cây

Bảng băm

Đồ thị

Text	Nhị phân	Mô tả
"r"	"rb"	Mở một file có sẵn để đọc
"w"	"wb"	Tạo file mới để ghi, nếu file có sẵn thì sẽ bị ghi mới hoàn toàn
"a"	"ab"	Mở một file để ghi từ vị trí cuối cùng của file, nếu file không tồn tại sẽ tạo mới
"r+"	"rb+"	Mở một file có sẵn để đọc và ghi
"w+"	"wb+"	Tạo file mới để đọc và ghi, nếu file có sẵn thì sẽ bị ghi mới hoàn toàn
"a+"	"ab+"	Mở một file để đọc và ghi. Có thể đọc từ đầu file, nhưng khi ghi thì ghi từ vị trí cuối cùng của file, nếu file không tồn tại sẽ tạo mới

Ôn tập xử lý file

c. Đóng kết nối:

`fclose(f); //đóng kết nối f đang có`

❖ Lưu ý:

- Chúng ta phải khai báo và tạo kết nối riêng.
- Mở file có thể thất bại, Khi đó f trả về giá trị NULL , do đó chúng ta phải kiểm tra f khác NULL thì mới thao tác trên file được.
- Khi thao tác xong trên file phải đóng file lại.

Cây

Bảng băm

Đồ thị

Ôn tập xử lý file

- Bảng ký tự chuyển đổi dạng thức:

%c : Ký tự đơn

%s : Chuỗi

%d : in số ra kiểu int

%f : in số ra kiểu float

%e : Số chấm động (ký hiệu có số mũ)

%g : Số chấm động (%f hay %g)

%u : Số nguyên thập phân không dấu(unsigned)

%x : in ra số ở dạng cơ số 16

%o : in ra số ở dạng cơ số 8

Cây

Bảng băm

Đồ thị

Ôn tập xử lý file

2. Thao tác đọc ghi trên file:

a. Cú pháp đọc:

➤ File text:

**fscanf_s(f,<ký tự chuyển đổi dạng thức>,<địa chỉ
của biến nhận giá trị đọc vào>);**

• ví dụ :

- chúng ta đọc 1 giá trị kiểu int vào biến n:

fscanf_s(f,"%d",&n);

Cây

Bảng băm

Đồ thị

Ôn tập xử lý file

❖ Lưu ý: Khi đọc kiểu chuỗi (%s):

- Trong cú pháp ta truyền tên biến chứ không phải địa chỉ biến.
- Cú pháp phải có thêm _countof
`fscanf_s(f, "%s", name, _countof(name));`
- Tuy nhiên %s chỉ đọc chuỗi không chứa khoảng trắng. Khi cần đọc một chuỗi có chứa khoảng trắng cú pháp có chút thay đổi như sau:

```
char name[120];
fscanf_s(f, "%120[^\\n]", name, _countof(name));
```

Cây

Bảng băm

Đồ thị

Ôn tập xử lý file

➤ File nhị phân

- Dùng lệnh **fread** để đọc.

- Cú pháp :

fread(<địa chỉ biến cần đọc>, <kích thước biến>, <đọc bao nhiêu lần>, <kết nối>);

- ví dụ :

fread(&n,sizeof(int),1,f);

đọc 1 biến n kiểu số nguyên từ file thông qua kết nối f.

Cây

Bảng băm

Đồ thị

Ôn tập xử lý file

b. Cú pháp ghi:

➤ File text:

Ta dùng **fprintf_s** để ghi file text.

fprintf_s(f,<ký tự chuyển đổi dạng thức, “\t hoặc \n nếu có” >,a1,a2,...,an);

- \n : xuống dòng.
- \t : canh tab.
- có bao nhiêu ký tự chuyển đổi dạng thức phải có bấy nhiêu biến a1, a2, .. kèm theo.

ví dụ: **fprintf_s(f,"%s \t %d \n",ten,tuoi);**

Cây

Bảng băm

Đồ thị

Ôn tập xử lý file

➤ File nhị phân

- Dùng lệnh **fwrite** để ghi đối với file nhị phân.
- Cú pháp :

fwrite(<địa chỉ biến cần ghi>, <kích thước biến>, <ghi bao nhiêu biến>, <kết nối>);

- ví dụ :

fwrite(&SvA, sizeof(SinhVien), 1, f);

ghi biến SvA kiểu SinhVien vào file thông qua kết nối f.

Cây

Bảng băm

Đồ thị

Ôn tập xử lý file

- Lưu ý: Kể từ VS2010 trở về sau, Các hàm fopen, fscanf, fprintf bị cảnh báo có thể phát sinh lỗi trong quá trình thực thi do đó được khuyến cáo không nên sử dụng. Để sử dụng được các hàm này có 2 cách xử lý sau:
 - Cách 1: thêm dòng sau vào đầu chương trình

```
#define _CRT_SECURE_NO_WARNINGS
```
 - Cách 2: Thay đổi Properties của project bằng GUI

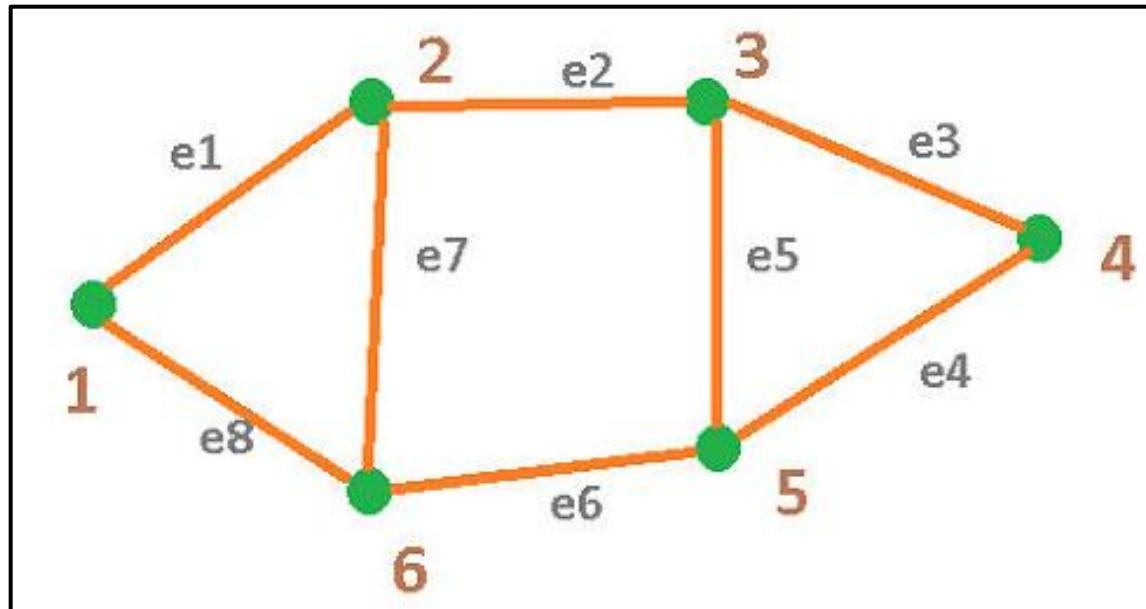
Nhấp phải tại tên Project chọn Properties → Configuration → C/C++ → Preprocessor → trong mục Preprocessos Definitions phía bên phải thêm vào dòng _CRT_SECURE_NO_WARNINGS → OK.

Cây

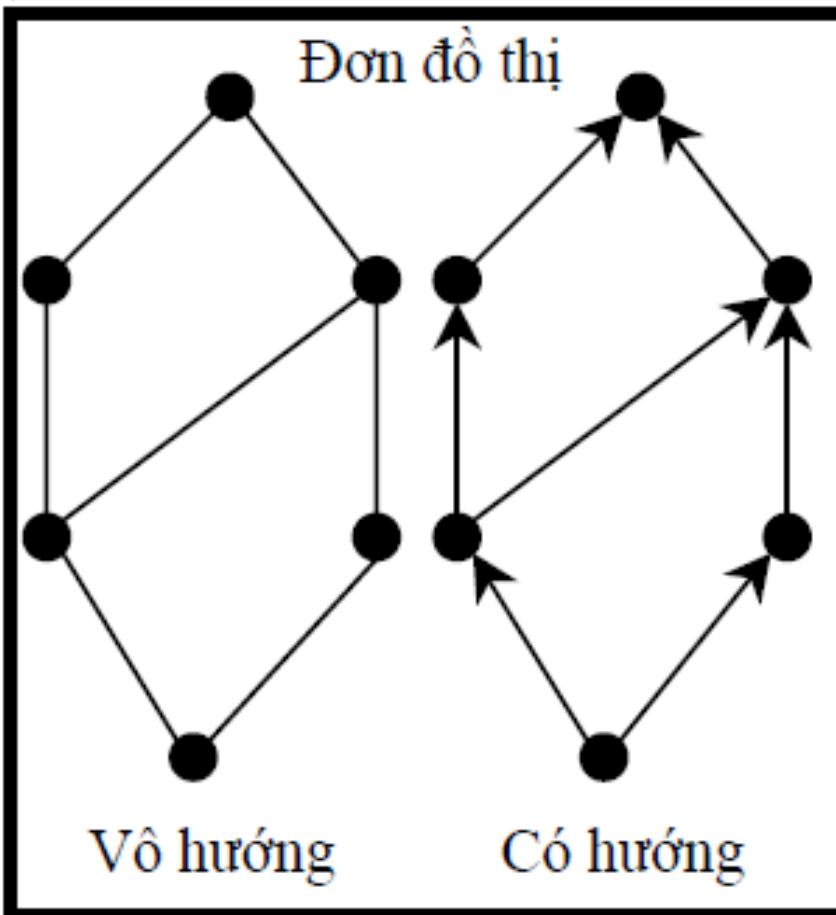
Bảng băm

Đồ thị

Đồ thị và các khái niệm cơ bản



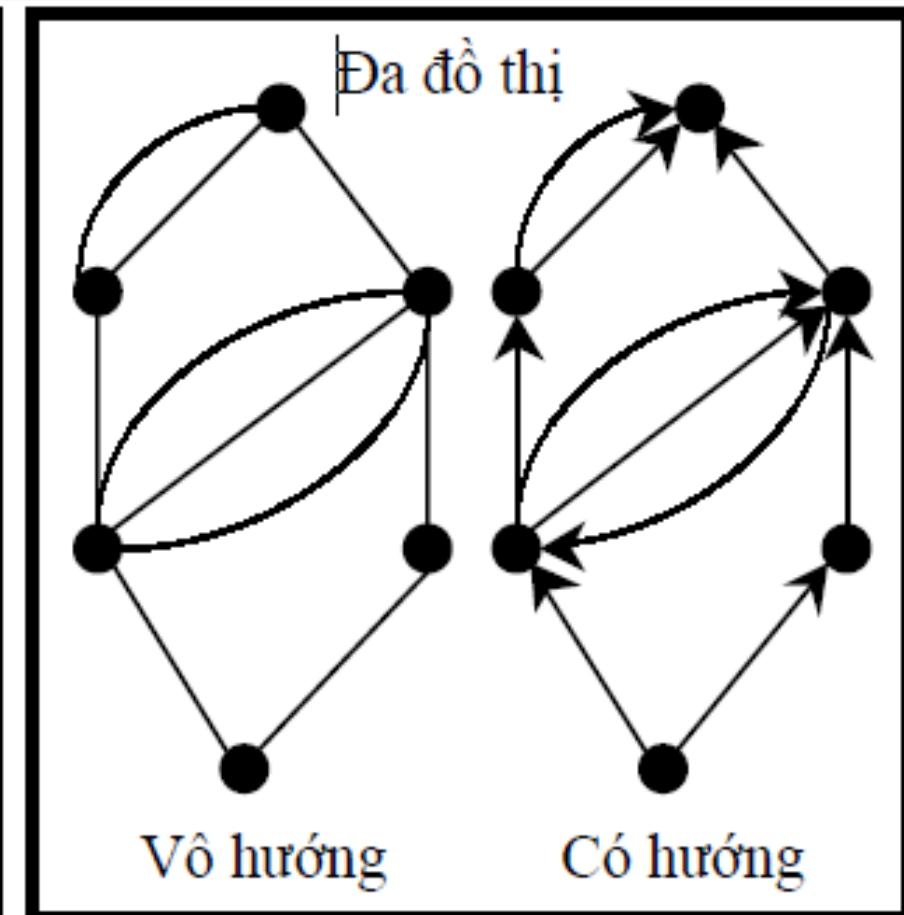
Đồ thị



Cây

Bảng băm

Đồ thị



Các khái niệm cơ bản

1. Khái niệm:

Đồ thị là một cấu trúc rời rạc gồm các đỉnh và các cạnh nối các đỉnh đó. Được mô tả :

G=(V,E) hoặc G(V,E)

- V được gọi là tập đỉnh (Verticies)
- E được gọi là tập cạnh (Edges)
- Với mọi $u, v \in V$ nếu $e = (u, v) \in E$ nghĩa là có cạnh nối từ đỉnh u đến đỉnh v trong G. Ta nói đỉnh u kề với đỉnh v. Cạnh e là cạnh kề(liên thuộc) với u và v.
- Hai cạnh nối cùng 1 cặp đỉnh là 2 cạnh song song
- Khi $u = v$ thì cạnh (u,v) được gọi là một khuyên.

Cây

Bảng băm

Đồ thị

Các khái niệm cơ bản

2. Phân loại đồ thị:

a. Dựa vào số lượng cạnh giữa hai đỉnh

- Đơn đồ thị
- Đa đồ thị

b. Dựa vào đặc tính của đồ thị:

- Đồ thị vô hướng
- Đồ thị có hướng (trong đồ thị có hướng thì 1 cạnh (u,v) được gọi là 1 cung đi từ u vào v , u được gọi là đỉnh đầu và v được gọi là đỉnh cuối).

Cây

Bảng băm

Đồ thị

Các khái niệm cơ bản

3. Bậc của một đỉnh trong đồ thị G(V,E):

➤ Khi G là đồ thị vô hướng, Bậc của đỉnh v, ký hiệu $\deg(v)$, là số cạnh kề với v, trong đó một khuyên tại một đỉnh được đếm 2 lần cho bậc của đỉnh ấy.

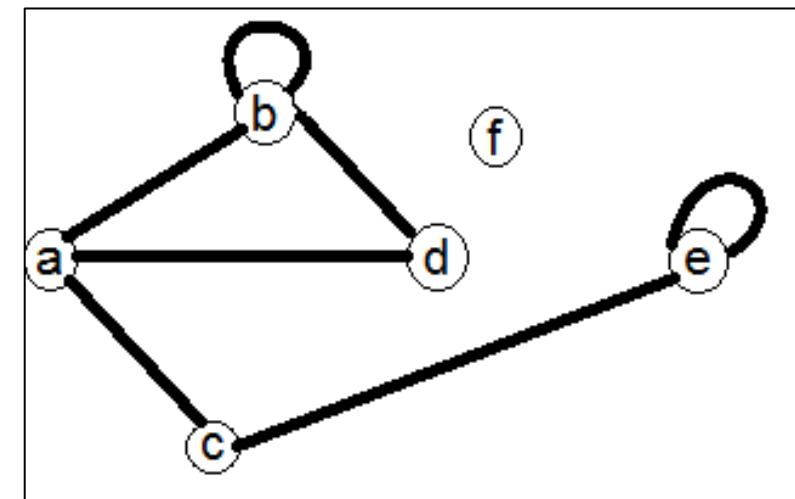
Ví dụ: $\deg(a)=3$

$\deg(b)=?$

$\deg(c)=?$

$\deg(e)=?$

$\deg(f)=?$



Các khái niệm cơ bản

➤ Khi G là đồ thị có hướng

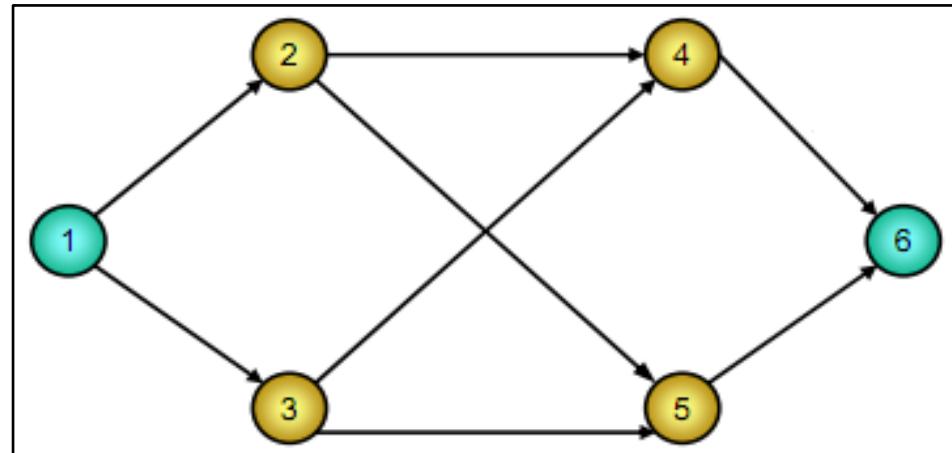
- $\text{deg}^-(v)$: số cung có đỉnh cuối là v , gọi là **bậc vào** của v
- $\text{deg}^+(v)$: số cung có đỉnh đầu là v , gọi là **bậc ra** của v
- $\text{deg}(v) = \text{deg}^-(v) + \text{deg}^+(v)$

$$\text{deg}^-(1) = 0$$

$$\text{deg}^+(1) = 2$$

$$\text{deg}^-(4) = ?$$

$$\text{deg}^+(4) = ?$$



➤ Đỉnh có bậc bằng 0 là đỉnh cô lập, đỉnh có bậc bằng 1 là đỉnh treo

Cây

Bảng băm

Đồ thị

Các khái niệm cơ bản

4. Định lí:

- Giả sử $G = (V, E)$ là đồ thị **vô hướng** với m cạnh, khi đó tổng tất cả các bậc đỉnh trong V sẽ bằng $2m$.
- Giả sử $G = (V, E)$ là đồ thị **có hướng** với m cạnh, khi đó tổng bậc vào của tất cả các đỉnh bằng tổng bậc ra của tất cả các đỉnh và bằng m
- Số đỉnh bậc lẻ của đồ thị là số chẵn.

Cây

Bảng băm

Đồ thị

Các khái niệm cơ bản

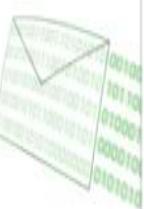
Bài tập:

- 1) Cho đồ thị G có 14 cạnh, trong đó có 3 đỉnh bậc 1, 2 đỉnh bậc 3, 2 đỉnh bậc 4, 1 đỉnh bậc 5, các đỉnh còn lại có bậc là 2. Hỏi G có bao nhiêu đỉnh?
- 2) Cho đồ thị G có 13 cạnh, trong đó có 3 đỉnh bậc 1, 4 đỉnh bậc 2, 1 đỉnh bậc 5, các đỉnh còn lại có bậc là 3 hoặc 4. Hỏi G có bao nhiêu đỉnh bậc 3 và đỉnh bậc 4?

Cây

Bảng băm

Đồ thị



Cây

Bảng băm

Đồ thị

Biểu diễn đồ thị trong máy tính

- Ma trận kề
 - Danh sách cạnh
 - Danh sách kề
- 

MA TRẬN KẾ

Biểu diễn đồ thị

1. Ma trận kề:

Giả sử $G = (V, E)$ là một **đơn đồ thị** có số đỉnh là $|V| = n$, các đỉnh được đánh số $1, 2, \dots, n$. Khi đó ta có thể biểu diễn đồ thị bằng một ma trận vuông $A = [a_{ij}]$ cấp n . Trong đó:

$$a_{ij} = 1 \text{ nếu } (i, j) \in E$$

$$a_{ij} = 0 \text{ nếu } (i, j) \notin E$$

$$a_{ii} = 0 \quad \forall i \in V$$

Đối với đa đồ thị thì việc biểu diễn cũng tương tự trên, nhưng $a_{ij} = k$, với k là số cạnh nối giữa 2 đỉnh i, j .

Cây

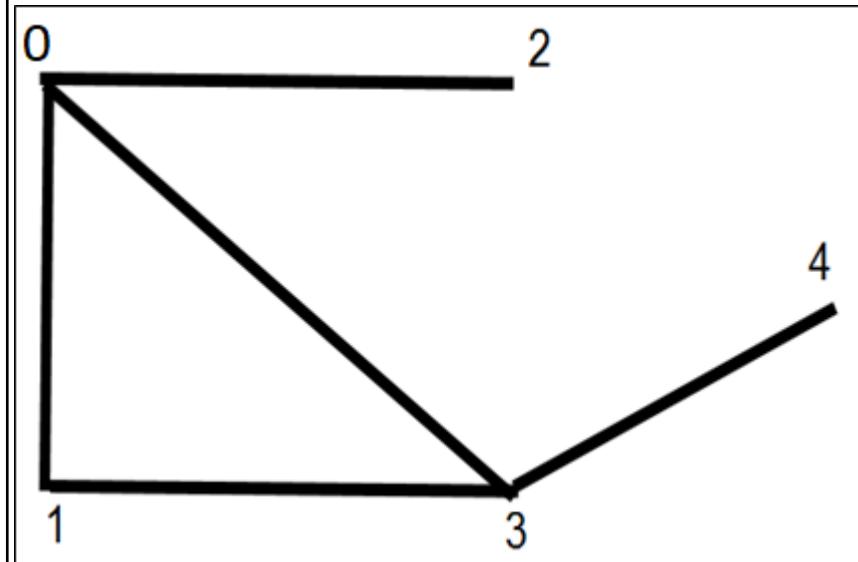
Bảng băm

Đồ thị

Biểu diễn đồ thị

Ma trận kề vô hướng:

		0	1	2	3	4
0	0	1	1	1	0	
1	1	0	0	1	0	
2	1	0	0	0	0	
3	1	1	0	0	1	
4	0	0	0	1	0	



Cây

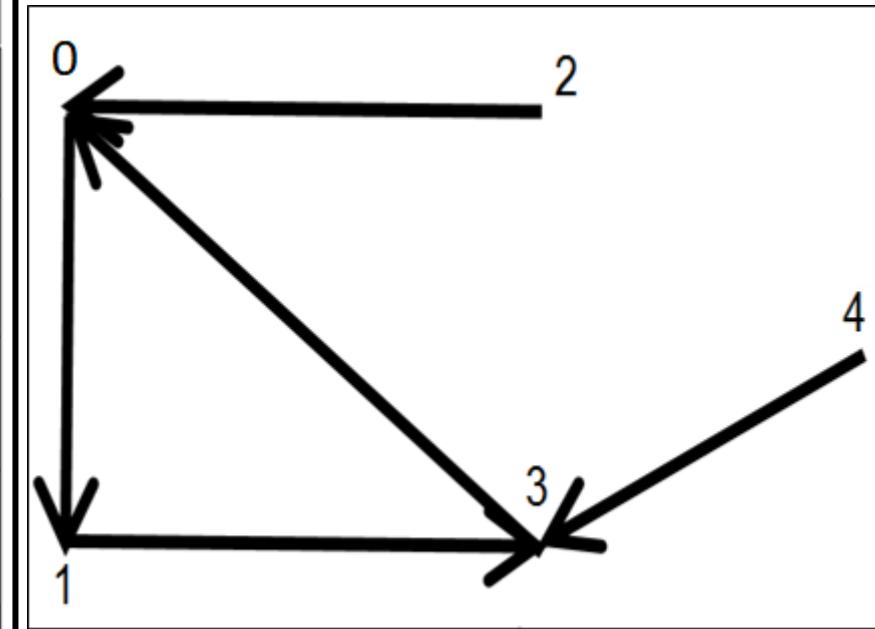
Bảng băm

Đồ thị

Biểu diễn đồ thị

Ma trận kề có hướng:

	0	1	2	3	4
0	0	1	0	0	0
1	0	0	0	1	0
2	1	0	0	0	0
3	1	0	0	0	0
4	0	0	0	1	0



Cây

Bảng băm

Đồ thị

Biểu diễn đồ thị

Các tính chất của ma trận kề:

Cho đồ thị G và A là ma trận kề của G thì:

- Nếu G vô hướng thì:
 - A là ma trận đối xứng.
 - Tổng các số trên hàng i = tổng các số trên cột i = bậc của đỉnh i = $\deg(i)$.
- Nếu G có hướng thì:
 - Tổng các số trên hàng i = bậc ra của đỉnh i = $\deg^+(i)$
 - Tổng các số trên cột i = bậc vào của đỉnh i = $\deg^-(i)$

Biểu diễn đồ thị

Cấu trúc của ma trận kề:

```
//MAXV: số đỉnh tối đa của đồ thị
const int MAXV=20;

//Ma trận kề của đồ thị (Adjacency-matrix)
struct AdjacencyMatrix
{
    int mt[MAXV][MAXV];
    int n;
};
```

Cây

Bảng băm

Đồ thị

Ưu nhược điểm của ma trận kề

Đồ thị

Ưu điểm:

- Đơn giản, trực quan, dễ cài đặt trên máy tính.
- Để kiểm tra xem hai đỉnh (u, v) của đồ thị có kề nhau hay không, ta chỉ việc kiểm tra bằng một phép so sánh: $a_{uv} \neq 0$.

Nhược điểm:

- Bất kể số cạnh của đồ thị n đỉnh, ma trận kề luôn luôn đòi hỏi n^2 ô nhớ để lưu các phần tử ma trận \Rightarrow tốn bộ nhớ và không biểu diễn được khi n lớn.
- Để duyệt qua tất cả các cạnh kề với đỉnh u ta luôn phải xét qua n phần tử trên dòng u của ma trận ngay cả khi u là đỉnh cô lập \Rightarrow tốn thời gian xử lý.

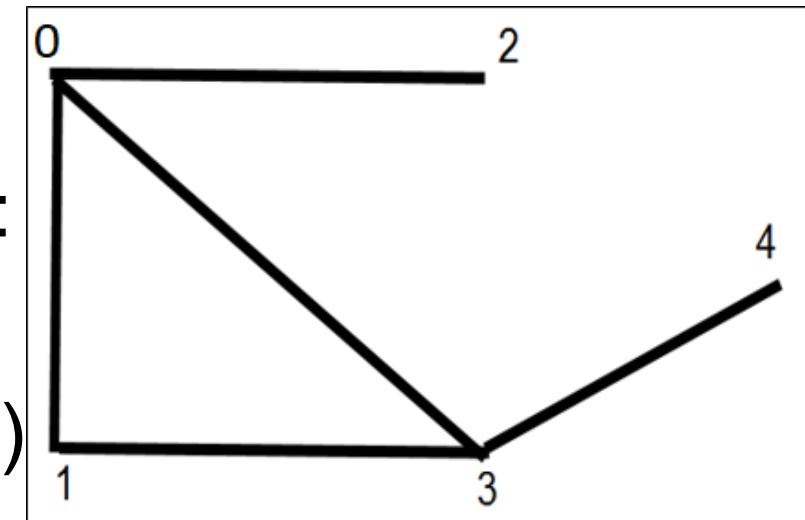
DANH SÁCH CẠNH

Biểu diễn đồ thị

2. Danh sách cạnh:

Giả sử $G = (V, E)$ là một đồ thị có n đỉnh và m cạnh. Ta có thể biểu diễn G bằng 1 biến chứa giá trị n và một danh sách chứa m cạnh của đồ thị.

Ví dụ với đồ thị bên cạnh:
 $n=5$, $m=5$ và danh sách
cạnh gồm $(0,1);(0,2);(0,3)$
 $(1,3);(3,4)$



Cây

Bảng băm

Đồ thị

Biểu diễn đồ thị

Cây

Bảng băm

Đồ thị

Cấu trúc danh sách cạnh (danh sách đặc):

```
//EDGE: Kiểu của một cạnh trong đồ thị
struct EDGE
{
    VERTEX org; //Origin
    VERTEX des; //Destination
};

//ArrayEdge: danh sách cạnh của đồ thị
struct ArrayEdge
{
    EDGE ds [MAXE];
    int count;
};
```

Số cạnh của đồ thị

Ưu nhược điểm của danh sách cạnh

Ưu điểm:

- Trong trường hợp đồ thị thưa (có số cạnh tương đối nhỏ: chẳng hạn $m < 6n$), cách biểu diễn bằng danh sách cạnh sẽ tiết kiệm được không gian lưu trữ.
- Trong một số trường hợp, ta phải xét tất cả các cạnh của đồ thị thì cài đặt trên danh sách cạnh làm cho việc duyệt các cạnh dễ dàng hơn. (Thuật toán Kruskal)

Nhược điểm: Khi ta cần duyệt tất cả các đỉnh kề với đỉnh v nào đó thì ta phải duyệt tất cả các cạnh, lọc ra những cạnh có chứa đỉnh v và xét đỉnh còn lại \Rightarrow khá tốn thời gian trong trường hợp đồ thị dày (nhiều cạnh).

Cây

Bảng băm

Đồ thị

DANH SÁCH KỀ

Biểu diễn đồ thị

3. Danh sách kề:

Để khắc phục những nhược điểm của ma trận kề và danh sách cạnh, người ta đề xuất một cấu trúc dữ liệu mới là danh sách kề. Trong cách biểu diễn này, với mỗi đỉnh v của đồ thị, ta cho tương ứng với nó một danh sách các đỉnh kề với v . Với đồ thị $G = (V, E)$. V gồm n đỉnh và E gồm m cạnh. Có hai cách cài đặt danh sách kề phổ biến:

- Cài đặt bằng danh sách đặc.
- Cài đặt bằng danh sách liên kết.

Cây

Bảng băm

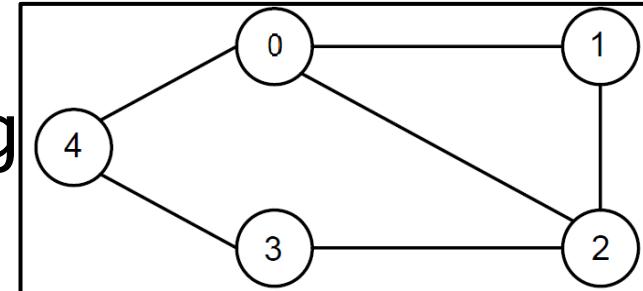
Đồ thị

Biểu diễn đồ thị

Cài đặc bằng danh sách đặc:

Dùng một mảng các đỉnh, mảng đó chia làm n đoạn, đoạn thứ i

trong mảng lưu danh sách các đỉnh kề với đỉnh i.



0	1	2	3	4	5	6	7	8	9	10	11
1	2	4	0	2	0	1	3	2	4	0	3
Đoạn 0		Đoạn 1		Đoạn 2		Đoạn 3		Đoạn 4			

Để biết một đoạn nằm từ chỉ số nào đến chỉ số nào, ta có một mảng lưu vị trí riêng. Ta gọi mảng lưu vị trí đó là mảng VT. VT[i] sẽ bằng chỉ số bắt đầu đoạn thứ i. VT=(0, 3, 5, 8, 10)

Cây

Bảng băm

Đồ thị

Biểu diễn đồ thị

Cài đặc bằng danh sách liên kết:

Dùng danh sách liên kết đơn, đồ thị G có n đỉnh nên ta xây dựng một danh sách đặc gồm n phần tử mà phần tử thứ i là 1 danh sách liên kết đơn chứa các đỉnh kề với đỉnh i. Với ví dụ trên thì danh sách kề của chúng ta là:

- DS[0]:

1	
---	--

 →

2	
---	--

 →

4	
---	--

 → NULL
- DS[1]:

0	
---	--

 →

2	
---	--

 → NULL
- DS[2]:

0	
---	--

 →

1	
---	--

 →

3	
---	--

 → NULL
- DS[3]:

2	
---	--

 →

4	
---	--

 → NULL
- DS[4]:

0	
---	--

 →

3	
---	--

 → NULL

Cây

Bảng băm

Đồ thị

Biểu diễn đồ thị

Cấu trúc danh sách kề (danh sách liên kết):

```
//Danh sách kề của đồ thị (Adjacency-list)
struct NodeVertex
{
    int ver; //Vertex
    NodeVertex* link;
};

typedef NodeVertex* NodeVertexPointer;

struct AdjacencyList
{
    NodeVertexPointer ds [MAXV];
    int count;
};
```

Cây

Bảng băm

Đồ thị

Ưu nhược điểm của danh sách kê

Ưu điểm:

- Việc duyệt tất cả các đỉnh kề với một đỉnh v cho trước là hết sức dễ dàng. Việc duyệt tất cả các cạnh cũng đơn giản vì một cạnh thực ra là nối một đỉnh với một đỉnh khác kề nó.

Nhược điểm:

- Về lý thuyết, so với hai phương pháp biểu diễn trên, danh sách kê tốt hơn hẳn. Tuy nhiên về mặt cài đặt danh sách kê có phần dài dòng hơn.

Cây

Bảng băm

Đồ thị



Cây

Bảng băm

Đồ thị

Duyệt đồ thị (BFS, DFS)



DUYỆT ĐỒ THỊ THEO CHIỀU RỘNG - BFS

Duyệt Breadth First Search

Cây

Bảng băm

Đồ thị

Ý tưởng của Breadth First Search (BFS):

Tìm kiếm theo chiều rộng (BFS) là một thuật toán tìm kiếm trong đồ thị bao gồm 2 thao tác:

- Cho trước một đỉnh của đồ thị.
- Thêm các đỉnh kề với đỉnh vừa cho vào danh sách có thể hướng tới tiếp theo.

Thuật toán BFS bắt đầu từ đỉnh gốc và lần lượt nhin các đỉnh kề với đỉnh gốc. Sau đó, với mỗi đỉnh trong số đó, thuật toán lại lần lượt nhin trước các đỉnh kề với nó mà chưa được quan sát trước đó và lặp lại.

Duyệt Breadth First Search

Ví dụ:

➤ Thứ tự duyệt từ đỉnh A là:

$A \Rightarrow B \Rightarrow D \Rightarrow C \Rightarrow E \Rightarrow F \Rightarrow G$

➤ Thứ tự duyệt từ đỉnh D là:

$D \Rightarrow A \Rightarrow B \Rightarrow F \Rightarrow C \Rightarrow E \Rightarrow G$

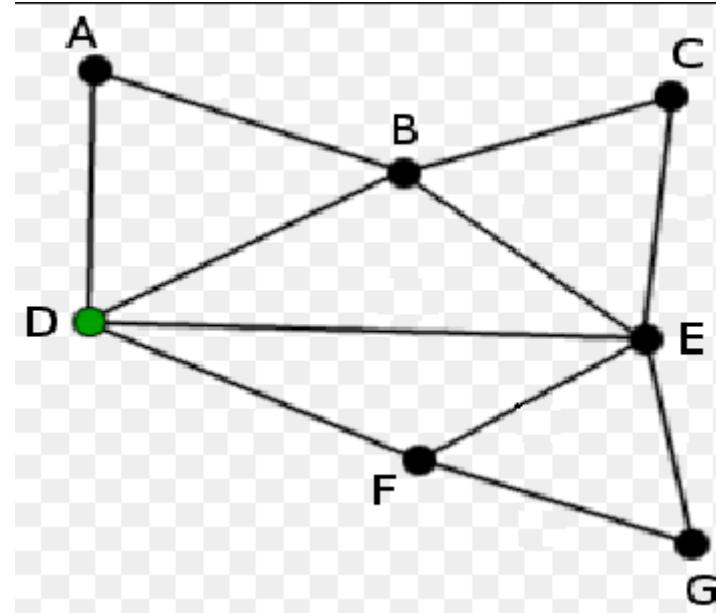
➤ Thứ tự **duyệt từ đỉnh G** là:

$G \Rightarrow ????$

Cây

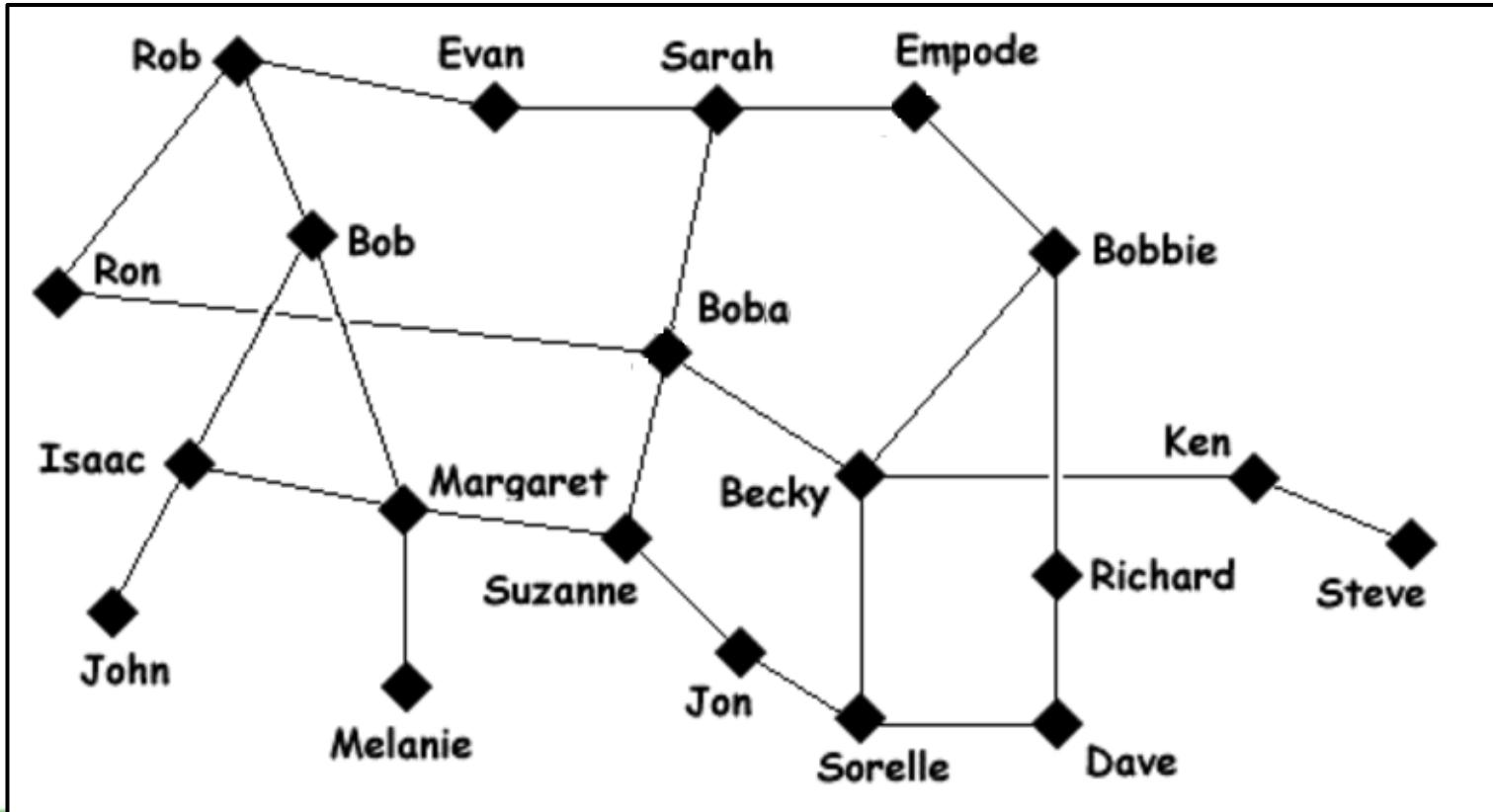
Bảng băm

Đồ thị



Duyệt Breadth First Search

Hãy mô tả quá trình duyệt theo chiều rộng đồ thị sau, xuất phát từ đỉnh Ron?



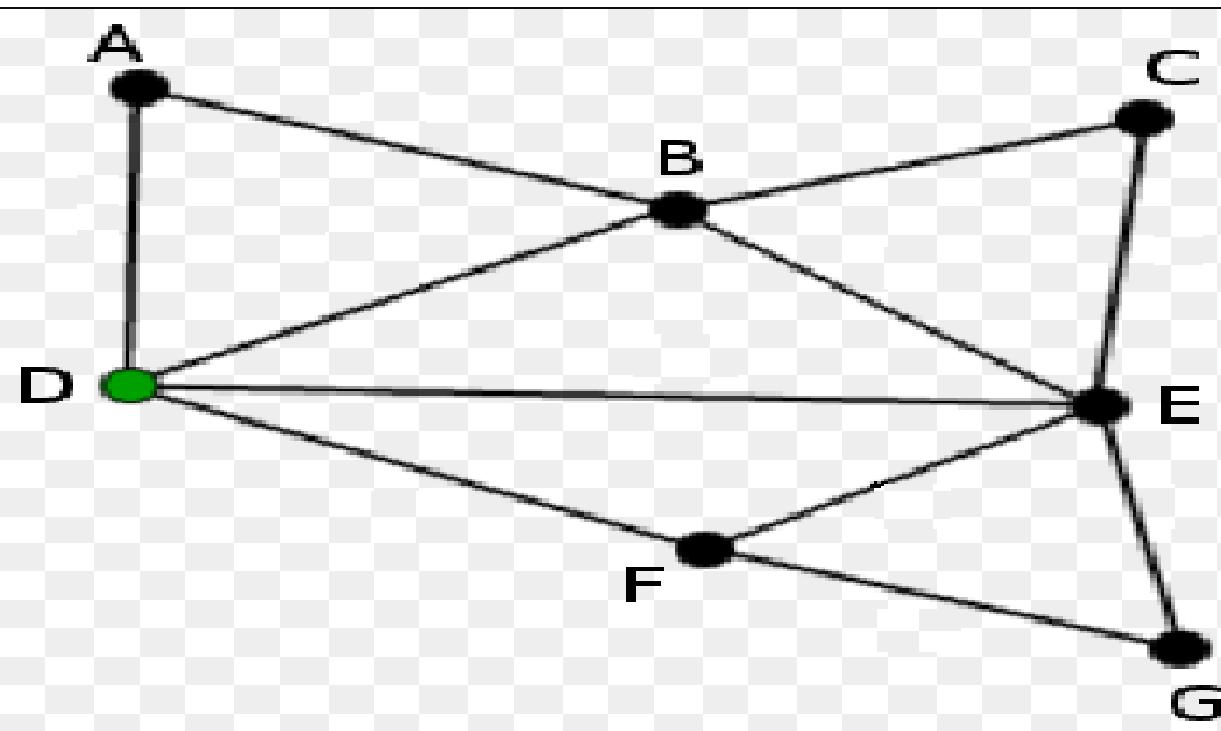
Cây

Bảng băm

Đồ thị

Duyệt Breadth First Search

Hãy mô tả quá trình duyệt theo chiều rộng đồ thị sau, xuất phát từ đỉnh A?



Cây

Bảng băm

Đồ thị

Duyệt Breadth First Search

Quá trình duyệt theo chiều rộng xuất phát từ đỉnh A:

Bước	Danh sách cần xét	Đỉnh đang xét	Đỉnh kề đỉnh đang xét mà chưa được xét
1	A	A	B, D
2	B, D	B	C, E
3	D, C, E	D	F
4	C, E, F	C	Ø
5	E, F	E	G
6	F, G	F	Ø
7	G	G	Ø
8	Ø		

Duyệt Breadth First Search

- ❖ Nhận xét rằng xuyên suốt quá trình thì sự thay đổi của danh sách cách đỉnh cần xét giống như việc chúng ta bỏ lần được các đỉnh cần xét và lấy chúng ra khỏi một **hàng đợi**.
- ❖ Do đó chúng ta sẽ xây dựng một thuật toán duyệt đồ thị theo chiều rộng sử dụng hàng đợi. Chúng ta sẽ cần một mảng kích thước bằng số đỉnh của đồ thị, phần tử thứ i của danh sách để đánh dấu xem đỉnh thứ i của đồ thị đã được quan sát hay chưa.

Cây

Bảng băm

Đồ thị

Duyệt Breadth First Search

Cấu trúc hàng đợi biểu diễn bằng danh sách đặc:

```
#define Max 100 //so phan tu toi da cua Queue
typedef int item; //kieu du lieu
struct Queue
{
    item Data[Max]; //Mang cac phan tu
    int count; //dem so phan tu cua Queue
};
```

Các phép toán trên hàng đợi:

```
void Init (Queue &Q); //khai tao Queue rong
int Isempty(Queue Q); //kiem tra Queue rong
int Isfull(Queue Q); //kiem tra Queue day
void Push(Queue &Q, item x); //them phan tu vao cuoi hang doi
int Pop(Queue &Q); //Loai bo phan tu khoi dau hang doi
int Qfront (Queue Q); //xem thong tin phan tu dau hang doi
```

Cây

Bảng băm

Đồ thị

Duyệt BFS xuất phát từ đỉnh S (sử dụng hàng đợi).

➤ Bước 1: khởi tạo:

- Các đỉnh đều chưa được đánh dấu ngoại trừ đỉnh xuất phát S.
- Một hàng đợi (Queue), ban đầu chỉ có một phần tử là S. Hàng đợi dùng để chứa các đỉnh sẽ được duyệt theo thứ tự ưu tiên chiều rộng.

➤ Bước 2: Lặp các bước sau đến khi hàng đợi rỗng.

- Lấy u khỏi hàng đợi, đánh dấu u (Bắt đầu việc duyệt đỉnh u)
- Xét tất cả những đỉnh v kề với u mà chưa được đánh dấu, với mỗi đỉnh v đó:
 - Đánh dấu v.
 - Ghi nhận vết đường đi từ u tới v (Có thể làm chung với việc đánh dấu)
 - Đẩy v vào hàng đợi (v sẽ chờ được duyệt tại những bước sau)

➤ Bước 3: Truy vết đường đi nếu cần.

Cây

Bảng băm

Đồ thị

□ DUYỆT ĐỒ THỊ THEO CHIỀU SÂU - DFS

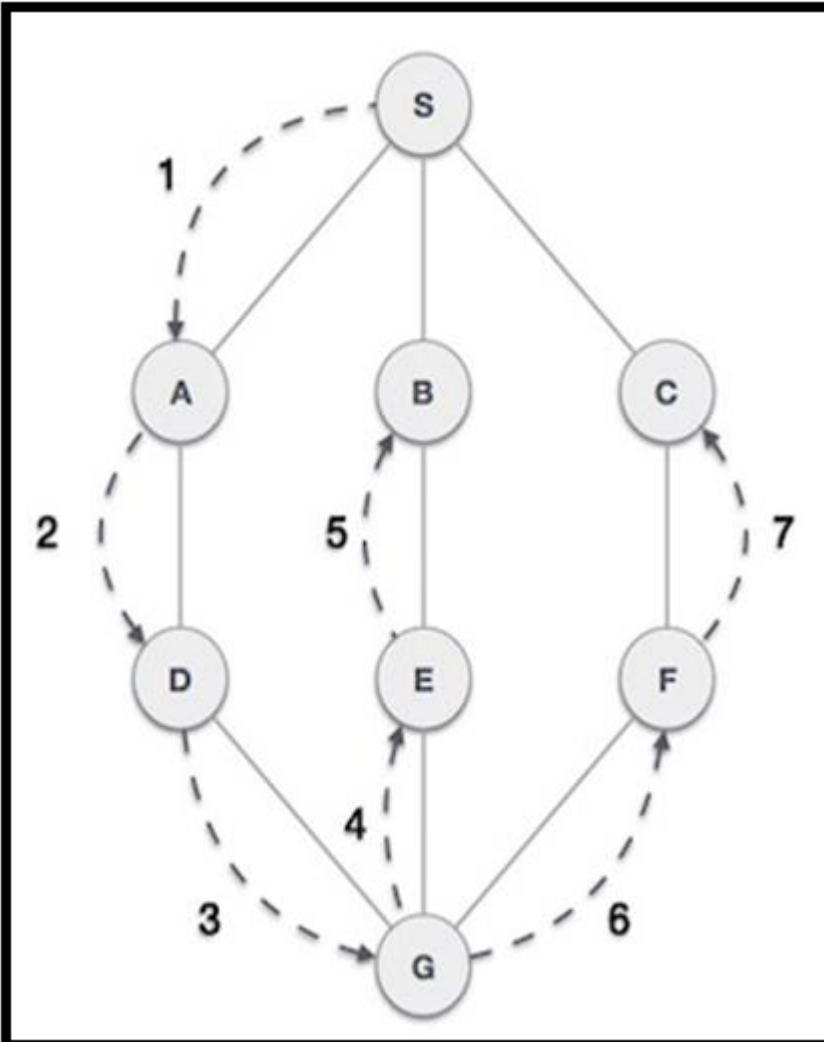
Duyệt Depth First Search

Ý tưởng của Depth First Search (DFS):

Tìm kiếm theo chiều sâu là thuật toán duyệt hoặc tìm kiếm trên một cây hoặc một đồ thị. Thuật toán khởi đầu tại gốc (hoặc chọn một đỉnh làm gốc) và phát triển xa nhất có thể theo mỗi nhánh.

Thông thường, DFS là một dạng thuật toán mà quá trình tìm kiếm được phát triển tới đỉnh con đầu tiên của nút đang tìm kiếm cho tới khi gặp được đỉnh cần tìm hoặc tới một nút không có con. Khi đó giải thuật quay lui về đỉnh vừa mới tìm kiếm ở bước trước.

Duyệt Depth First Search



Thứ tự duyệt theo chiều sâu
xuất phát từ đỉnh S

Cây

Bảng băm

Đồ thị

Duyệt Depth First Search

Giải thuật đệ quy (DFS) xuất phát từ đỉnh S:

Rõ ràng mọi đỉnh x kề với S tất nhiên sẽ đến được từ S và với mọi đỉnh x đó thì các đỉnh y kề với nó cũng đến được từ S. Điều đó gợi ý cho ta viết một thủ tục đệ quy DFS(u) mô tả việc duyệt từ đỉnh u bằng cách thông báo thăm đỉnh u và tiếp tục quá trình duyệt DFS(v) với v là một đỉnh chưa thăm kề với u . Để không một đỉnh nào bị liệt kê tới hai lần, ta sử dụng kỹ thuật đánh dấu, mỗi lần thăm một đỉnh, ta đánh dấu đỉnh đó lại để các bước duyệt đệ quy kế tiếp không duyệt lại đỉnh đó nữa.

Cây

Bảng băm

Đồ thị

Duyệt Depth First Search

Khởi tạo mảng đánh dấu(ban đầu các đỉnh đều chưa được duyệt).

Giải thuật đệ quy (DFS) xuất phát từ đỉnh S:

B1: thông báo đến được S

B2: đánh dấu S

B3: với tất cả những đỉnh **u** kề với S mà chưa được đánh dấu ta đệ quy **DFS(u)**.

B4: truy vết đường đi nếu cần.

Cây

Bảng băm

Đồ thị

Duyệt Depth First Search

❖ Nhận xét:

- Vì có kỹ thuật đánh dấu, nên thủ tục DFS sẽ được gọi $\leq n$ lần (n là số đỉnh)
- Đường đi từ S tới F có thể có nhiều, ở trên chỉ là một trong số các đường đi. Cụ thể là đường đi có thứ tự từ điển nhỏ nhất.
- Chúng ta có thể sử dụng mảng VISIT để lưu vết đường đi như đã đề cập trong BFS, VISIT xem như là 1 tham số của hàm.

Cây

Bảng băm

Đồ thị

Duyệt Depth First Search

Cấu trúc stack biểu diễn bằng danh sách đặc:

```
#define Max 100 //so phan tu toi da cua Stack
typedef int item; //kieu du lieu cua Stack
struct Stack
{
    int Top; //Dinh Top
    item Data[Max]; //Mang cac phan tu
};
```

Các phép toán trên stack:

```
void Init (Stack &S); //khoi tao Stack rong
int Iseempty(Stack S); //kiem tra Stack rong
int Isfull(Stack S); //kiem tra Stack day
void Push(Stack &S, item x); //them phan tu vao Stack
int Peak(Stack S); //Lay phan tu dau Stack nhung k xoa
int Pop(Stack &S); //Loai bo phan tu khoi Stack
```

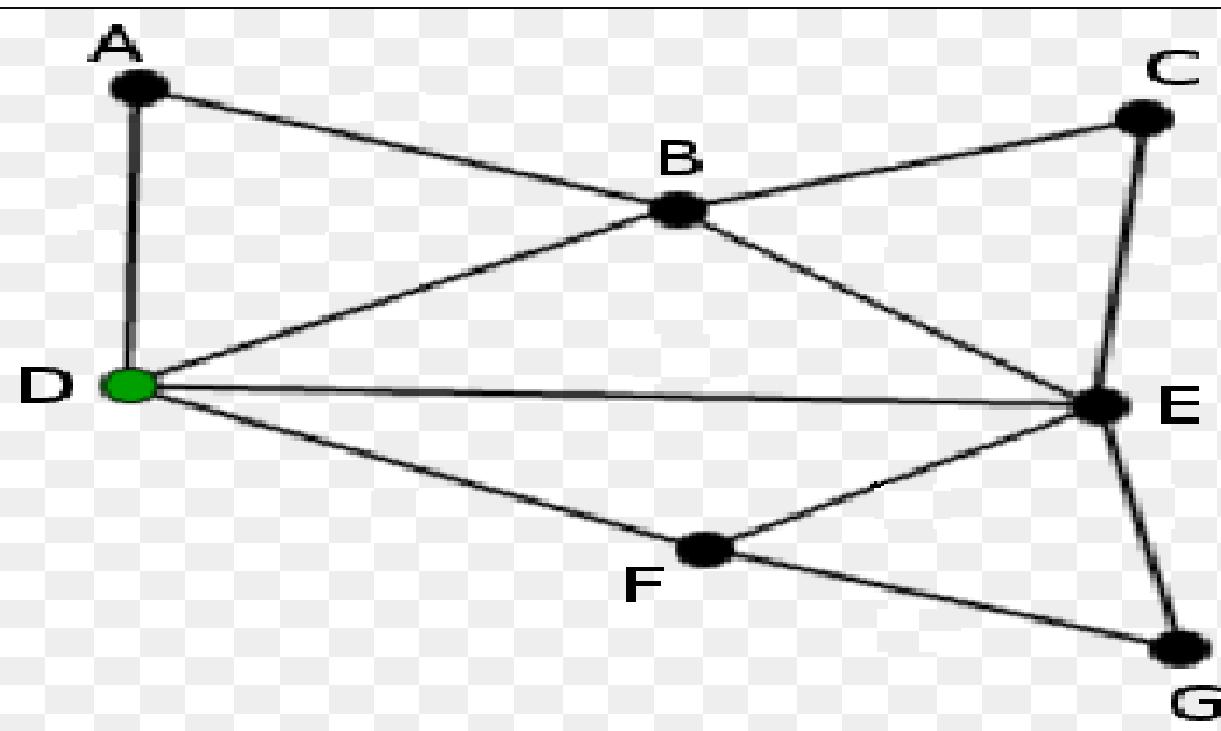
Cây

Bảng băm

Đồ thị

Duyệt Depth First Search

Hãy mô tả quá trình duyệt theo chiều sâu đồ thị sau, xuất phát từ đỉnh A?



Cây

Bảng băm

Đồ thị

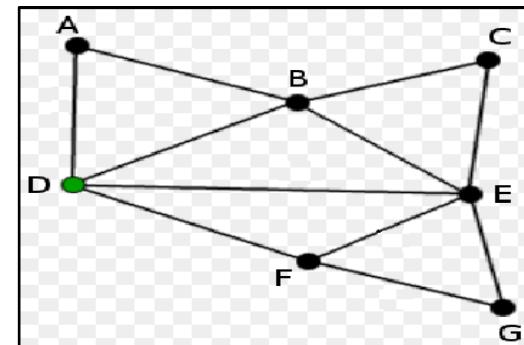
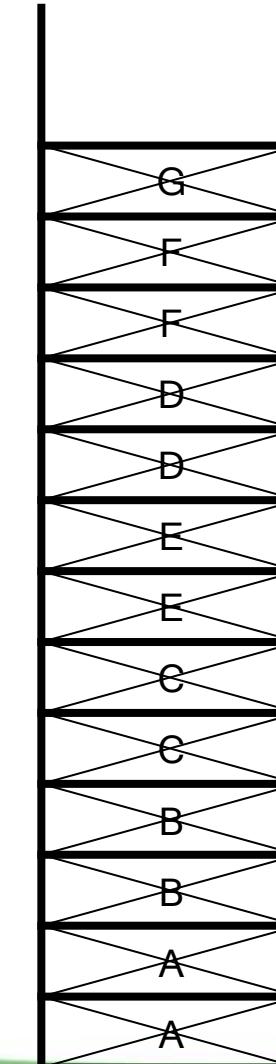
Duyệt Depth First Search

Bước	Stack		Đánh dấu
	Lấy ra	Bỏ vào	
Khởi tạo		A	A
1	A	A,B	B
2	B	B,C	C
3	C	C,E	E
4	E	E,D	D
5	D	D,F	F
6	F	F,G	G
7	G		
8	F		
9	D		
10	E		
11	C		
12	B		
13	A		

Cây

Bảng băm

Đồ thị



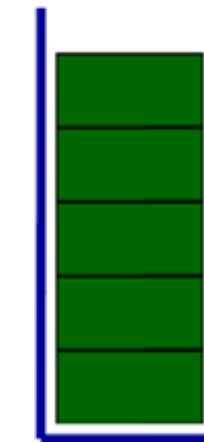
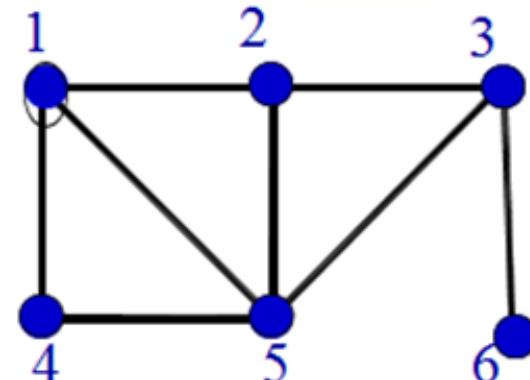
Duyệt Depth First Search

Hãy mô tả quá trình
Duyệt đồ thị bên theo
chiều sâu xuất phát từ
đỉnh số 1

Cây

Bảng băm

Đồ thị



Stack

Thứ tự duyệt:



Duyệt Depth First Search

Duyệt DFS không đê quy xuất phát từ đỉnh S:

B1: khởi tạo 1 mảng VISIT, 1 stack rỗng, thêm S vào stack, đánh dấu S đã viếng thăm

B2: (bước lặp) Trong khi stack khác rỗng:

- Lấy u ra khỏi stack
- Nếu u có đỉnh kề chưa thăm
 - + Chỉ chọn 1 đỉnh v kề u chưa được thăm.
 - + Đánh dấu thăm v
 - + Đẩy u vào lại stack (**lưu lại bước lùi**)
 - + Đẩy v vào stack

B3: truy vết đường đi nếu cần

Cây

Bảng băm

Đồ thị



Cây

Bảng băm

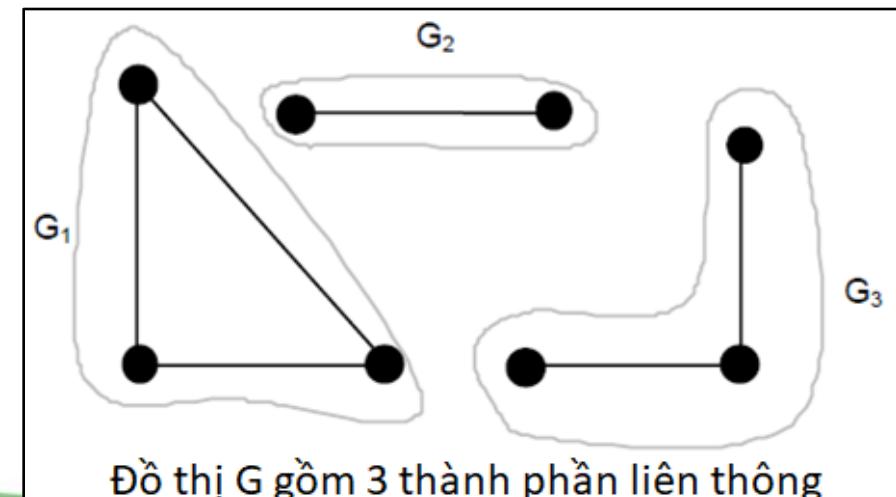
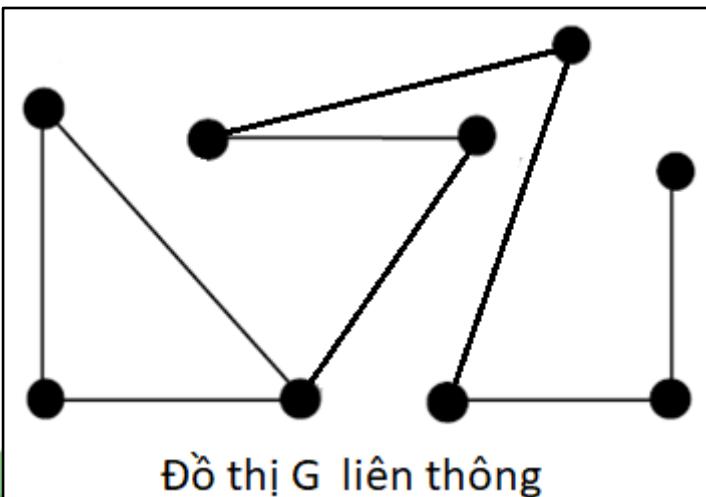
Đồ thị

Đồ thị liên thông

Đồ thị liên thông

Đối với đồ thị vô hướng $G = (V, E)$

G gọi là liên thông (connected) nếu luôn tồn tại đường đi giữa mọi cặp đỉnh phân biệt của đồ thị. Nếu G không liên thông thì chắc chắn nó sẽ là hợp của hai hay nhiều đồ thị con liên thông, các đồ thị con này đôi một không có đỉnh chung. Các đồ thị con liên thông rời nhau như vậy được gọi là các thành phần liên thông của đồ thị đang xét.



Cây

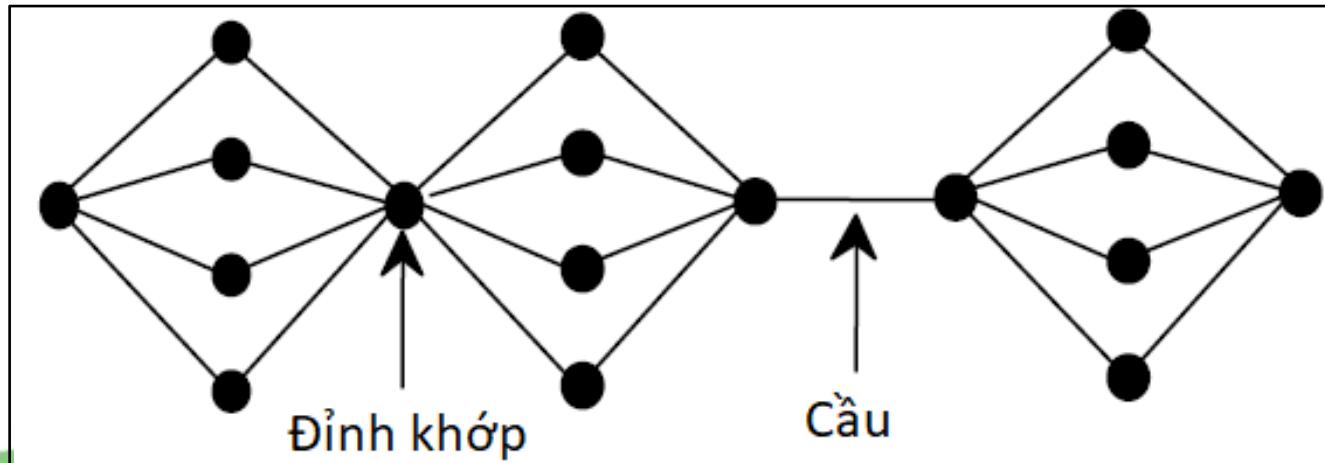
Bảng băm

Đồ thị

Đồ thị liên thông

➤ Đỉnh khớp và cầu.

Đôi khi, việc xoá đi một đỉnh và tất cả các cạnh liên thuộc với nó sẽ tạo ra một đồ thị con mới có nhiều thành phần liên thông hơn đồ thị ban đầu, các đỉnh như thế gọi là đỉnh cắt hay điểm khớp. Hoàn toàn tương tự, những cạnh mà khi ta bỏ nó đi sẽ tạo ra một đồ thị có nhiều thành phần liên thông hơn so với đồ thị ban đầu được gọi là một cạnh cắt hay một cầu.



Cây

Bảng băm

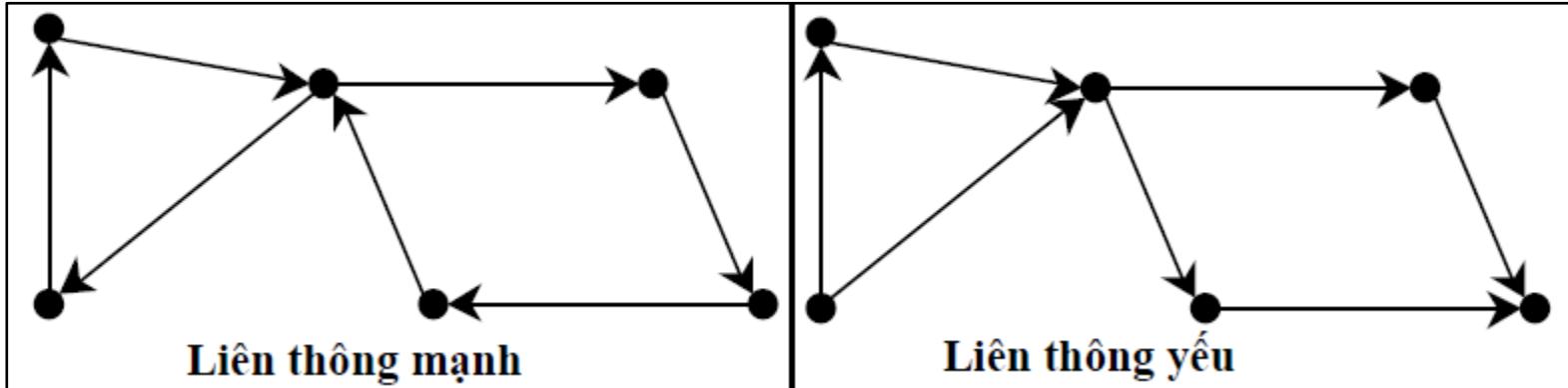
Đồ thị

Đồ thị liên thông

Đối với đồ thị có hướng $G = (V, E)$

Có hai khái niệm về tính liên thông của đồ thị có hướng tùy theo chúng ta có quan tâm tới hướng của các cung không.

G gọi là **liên thông mạnh** (Strongly connected) nếu luôn tồn tại đường đi (theo các cung định hướng) giữa hai đỉnh bất kỳ của đồ thị, g gọi là **liên thông yếu** (weakly connected) nếu đồ thị vô hướng nền của nó là liên thông.



Cây

Bảng băm

Đồ thị

Đồ thị liên thông

Bài toán:

- Viết hàm kiểm tra tính liên thông của đồ thị vô hướng.
- Kiểm tra 1 đồ thị có hướng có liên thông mạnh hay không
- Đếm số thành phần liên thông của đồ thị.

Gợi ý: Có thể dùng một mảng đánh dấu truyền vào hàm duyệt đồ thị. Sử dụng kết quả đánh dấu để giải quyết các bài toán này.

Cây

Bảng băm

Đồ thị



Cây

Bảng băm

Đồ thị

Sắp xếp topo

Sắp xếp topo

Cho $G = (V, E)$ là đồ thị vô hướng $u, v \in V$
Đường đi (dây chuyền) có chiều dài k nối hai
đỉnh u, v là dây đỉnh và cạnh liên tiếp nhau
 $v_0 e_1 v_1 e_2 \dots v_{k-1} e_k v_k$ sao cho:

$$v_0 = u, v_k = v, e_i = v_{i-1} v_i, i = 1, 2, \dots, k$$

Đường đi AG:

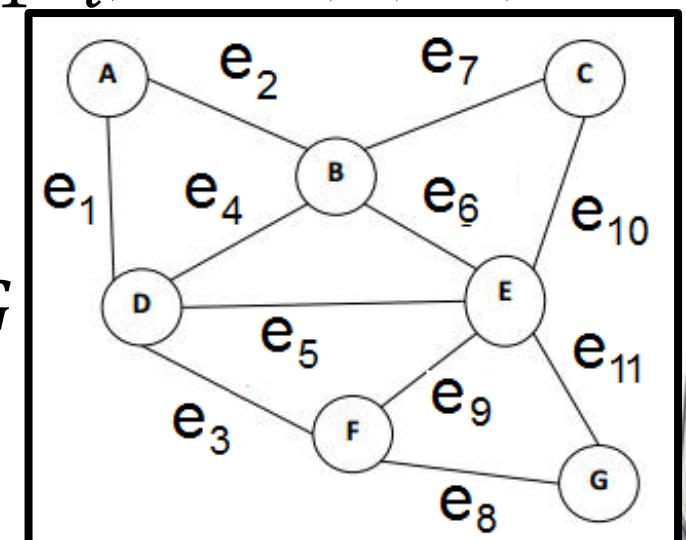
- $Ae_1De_5Ee_{11}G$
- $Ae_2Be_7Ce_{10}Ee_5De_3Fe_9Ee_{11}G$

Đường đi BG??

Cây

Bảng băm

Đồ thị



Sắp xếp topo

- Đường đi không có cạnh nào xuất hiện quá 1 lần gọi là đường đi đơn.
- Đường đi không có đỉnh nào xuất hiện quá một lần gọi là đường đi sơ cấp.
- Đường đi được gọi là chu trình nếu nó bắt đầu và kết thúc tại cùng một đỉnh.
- Đường đi được gọi là chu trình sơ cấp nếu nó bắt đầu và kết thúc tại cùng một đỉnh và không có đỉnh nào xuất hiện quá một lần.

Cây

Bảng băm

Đồ thị

Sắp xếp topo

Trong khoa học máy tính, thứ tự tô pô của một **đô thị có hướng** là một thứ tự sắp xếp của các đỉnh sao cho với mọi cung từ u đến v trong đồ thị, u luôn nằm trước v. Thuật toán để tìm thứ tự tô pô gọi là thuật toán sắp xếp tô pô. Thứ tự tô pô tồn tại khi và chỉ khi đồ thị không có chu trình. Đô thị có hướng không có chu trình luôn có ít nhất một thứ tự tô pô, và có thuật toán để tìm thứ tự tô pô trong thời gian tuyến tính.

Cây

Bảng băm

Đồ thị

Sắp xếp topo

B1: khởi tạo(L, S có thể dùng hàng đợi)

- $L \leftarrow$ danh sách rỗng (sẽ chứa danh sách đã sắp xếp)
- $S \leftarrow$ tập hợp các nút không có cung vào

B2: Bước lặp

while S khác rỗng

- Loại bỏ một nút n từ S .
- Chèn n vào L
- Với mỗi nút m sao cho có cung e từ n đến m .
 - Loại bỏ cung e từ đồ thị.
 - Nếu m không có cung vào thì chèn m vào S .

B3: xuất kết quả

Nếu đồ thị vẫn còn cung thì thông báo lỗi (đồ thị có ít nhất một chu trình) ngược lại thông báo thứ tự tô pô là L

Cây

Bảng băm

Đồ thị



Cây

Bảng băm

Đồ thị

Đường đi ngắn nhất (Floyd, Dijkstra)

ĐỒ THỊ TRỌNG SỐ

Đồ thị có trọng số

Đồ thị mà mỗi cạnh của nó được gán cho tương ứng với một số (nguyên hoặc thực) được gọi là đồ thị có trọng số.

Số gán cho mỗi cạnh của đồ thị được gọi là trọng số của cạnh.

Tương tự như đồ thị không trọng số, có nhiều cách biểu diễn đồ thị có trọng số trong máy tính như ma trận kề, danh sách cạnh hay danh sách kề. Đối với đơn đồ thị thì cách dễ dùng nhất là sử dụng ma trận trọng số.

Cây

Bảng băm

Đồ thị

Ma trận trọng số

Giả sử đồ thị $G = (V, E)$ có n đỉnh. Ta xây dựng ma trận vuông c kích thước $n \times n$:

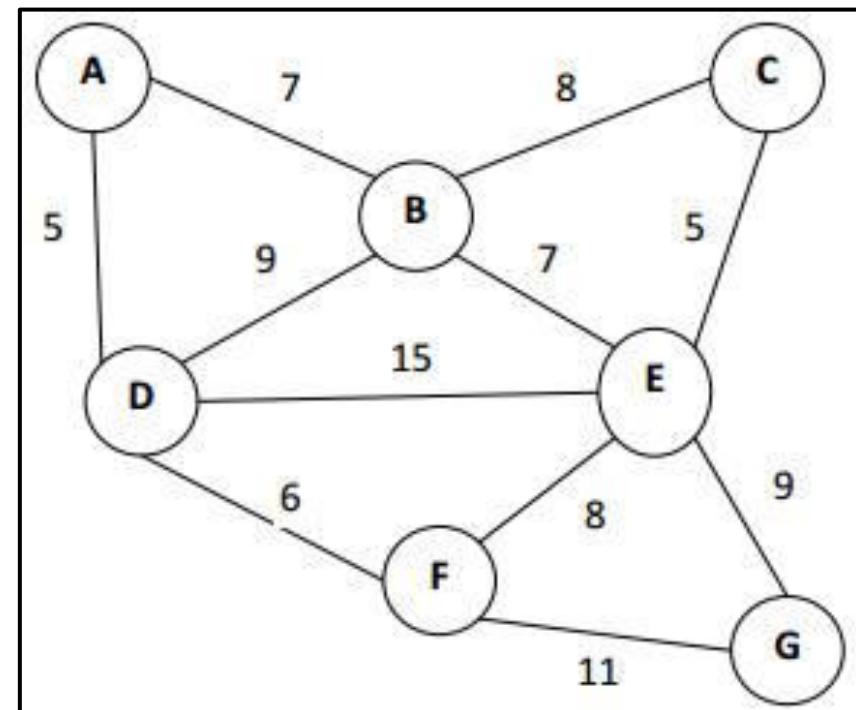
- Nếu $(u, v) \in E$ thì $C[u, v] = \text{trọng số của cạnh } (u, v)$.
- Nếu $(u, v) \notin E$ thì tùy theo trường hợp cụ thể, $C[u, v]$ được gán một giá trị nào đó để có thể nhận biết được (u, v) không phải là cạnh.
- Quy ước $c[v, v] = 0$ với mọi đỉnh v .

Đường đi, chu trình trong đồ thị có trọng số cũng được định nghĩa giống như trong trường hợp không trọng số, tuy nhiên độ dài đường đi không phải tính bằng số cạnh đi qua, mà được tính bằng tổng trọng số của các cạnh đi qua.

Ma trận trọng số của đồ thị

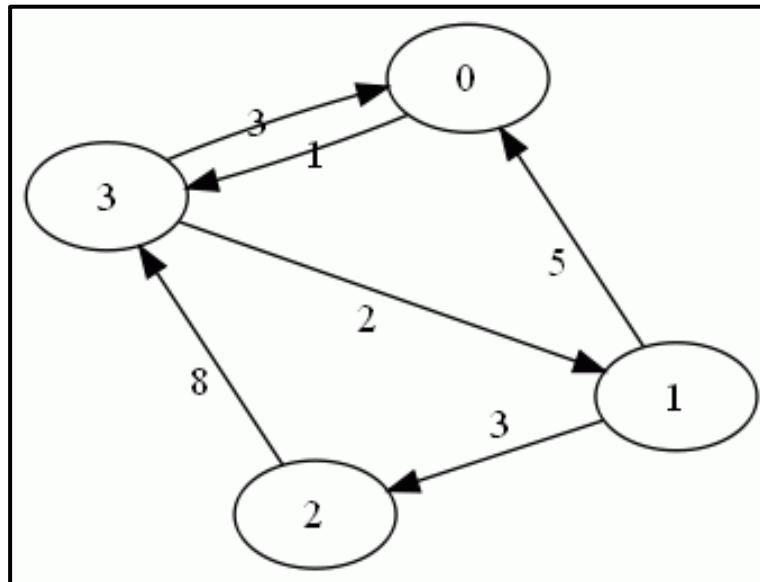
Ma trận trọng số của đồ thị là:

	A	B	C	D	E	F	G
A	0	7	999	5	999	999	999
B	7	0	8	9	7	999	999
C	999	8	0	999	5	999	999
D	5	9	999	0	15	6	999
E	999	7	5	15	0	8	9
F	999	999	999	6	8	0	11
G	999	999	999	999	9	11	0



Ma trận trọng số

Viết ma trận trọng số cho đồ thị sau:



Cây

Bảng băm

Đồ thị

BÀI TOÁN ĐƯỜNG ĐI NGĂN NHẤT

Bài toán đường đi ngắn nhất

Trong các App thực tế, chẳng hạn trong mạng lưới giao thông (Uber, Grab,...) người ta không chỉ quan tâm đến việc tìm đường đi giữa hai địa điểm mà còn phải lựa chọn một hành trình tiết kiệm nhất (theo tiêu chuẩn không gian, thời gian hay chi phí). Khi đó phát sinh yêu cầu tìm đường đi ngắn nhất giữa hai đỉnh của đồ thị. Bài toán đó phát biểu dưới dạng tổng quát: **Cho đồ thị trọng số $G = (V, E)$, hãy tìm đường đi ngắn nhất từ đỉnh S đến đỉnh F . Độ dài của đường đi này ta sẽ ký hiệu là $d[S, F]$. Nếu như không tồn tại đường đi từ S tới F thì khoảng cách đó = $+\infty$.**

Bài toán đường đi ngắn nhất

Nếu như đồ thị có chu trình âm (chu trình với độ dài âm) thì khoảng cách giữa một số cặp đỉnh nào đó có thể không xác định, bởi vì bằng cách đi vòng theo chu trình này một số lần đủ lớn, ta có thể chỉ ra đường đi giữa hai đỉnh nào đó trong chu trình này nhỏ hơn bất kỳ một số cho trước nào. Trong trường hợp như vậy, có thể đặt vấn đề tìm **đường đi cơ bản** (đường đi không có đỉnh lặp lại) ngắn nhất. Vấn đề đó là một vấn đề hết sức phức tạp mà ta sẽ không bàn tới ở đây. Trong nội dung chương trình ta chỉ xét đồ thị không có chu trình âm.

Bài toán đường đi ngắn nhất

Nếu như đồ thị không có chu trình âm thì ta có thể chứng minh được rằng một trong những đường đi ngắn nhất là đường đi cơ bản. Và nếu như biết được khoảng cách từ S tới tất cả những đỉnh khác thì đường đi ngắn nhất từ S tới F có thể tìm được một cách dễ dàng qua các thuật toán sau:

- Tìm đường đi ngắn nhất giữa mọi cặp đỉnh trong đồ thị: thuật toán Floyd.
- Tìm đường đi ngắn nhất từ 1 đỉnh đến tất cả các đỉnh còn lại Thuật toán Dijkstra.

Cây

Bảng băm

Đồ thị

□ THUẬT TOÁN FLOYD

Thuật toán Floyd

Bài toán: Cho đơn đồ thị có hướng, có trọng số $G = (V, E)$ với n đỉnh và m cạnh. Hãy tính tất cả các $d(u, v)$ là khoảng cách từ u tới v .

➤ Tư tưởng chính của thuật toán là để đi trực tiếp từ a đến b chúng ta mất 1 quãng đường là x . Thuật toán sẽ tìm 1 đường đi gián tiếp từ a qua k rồi đến b và nếu đường đi này ngắn hơn đường đi trực tiếp thì ta gán luôn giá trị nhỏ nhất của đường đi trực tiếp bằng đường đi gián tiếp.

Cây

Bảng băm

Đồ thị

Thuật toán Floyd

B1: khởi tạo $c' = c$. $\text{Trace}[u][v]=v$ (mọi đường đi ban đầu đều trực tiếp)

B2: Tính lại các $c'[u, v]$ thành độ dài đường đi ngắn nhất từ u tới v thông qua các đỉnh khác:

- Với mọi đỉnh k của đồ thị xét theo thứ tự từ 0 tới $n - 1$
 - Xét mọi cặp đỉnh u, v .
 - Nếu $c[u, v] > c[u, k] + c[k, v]$
 - $c[u, v] = c[u, k] + c[k, v]$
 - $\text{Trac}[u, v] = \text{Trac}[u, k]$ (Lưu lại vết để tìm đường)

B3: Truy vết dựa vào ma trận Trace nếu cần.
($\text{Trace}[u][v]=k$ nghĩa là k là đỉnh liền sau u trên đường đi từ u đến v)

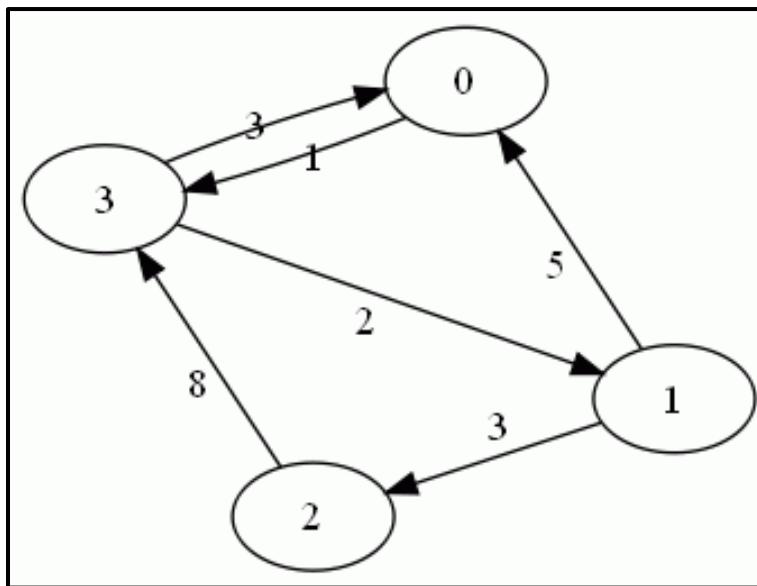
Cây

Bảng băm

Đồ thị

Ma trận trọng số

Demo: quá trình tìm đường đi ngắn nhất giữa tất cả các cặp đỉnh.



	0	1	2	3
0	0	999	999	1
1	5	0	3	999
2	999	999	0	8
3	3	2	999	0

Ma trận trọng số

C'

0	1	2	3	
0	0	999	999	1
1	5	0	3	999
2	999	999	0	8
3	3	2	999	0

Cây

Bảng băm

Đồ thị

Khởi tạo

Trace

0	1	2	3	
0	0	1	2	3
1	0	1	2	3
2	0	1	2	3
3	0	1	2	3

Ma trận trọng số

Đỉnh 0 làm đỉnh trung gian (dòng 0, cột 0 và chéo chính không đổi)

C'

	0	1	2	3
0	0	999	999	1
1	5	0	3	6
2	999	999	0	8
3	3	2	999	0

Trace

	0	1	2	3
0	0	1	2	3
1	0	1	2	0
2	0	1	2	3
3	0	1	2	3

Ma trận trọng số

Đỉnh 1 làm đỉnh trung gian (dòng 1, cột 1 và chéo chính không đổi)

C'

	0	1	2	3
0	0	999	999	1
1	5	0	3	6
2	999	999	0	8
3	3	2	5	0

Trace

	0	1	2	3
0	0	1	2	3
1	0	1	2	0
2	0	1	2	3
3	0	1	1	3

Ma trận trọng số

Đỉnh 2 làm đỉnh trung gian (dòng 2, cột 2 và chéo chính không đổi)

C'

	0	1	2	3
0	0	999	999	1
1	5	0	3	6
2	999	999	0	8
3	3	2	5	0

Trace

	0	1	2	3
0	0	1	2	3
1	0	1	2	0
2	0	1	2	3
3	0	1	1	3

Ma trận trọng số

Đỉnh 3 làm đỉnh trung gian (dòng 3, cột 3 và chéo chính không đổi)

C'

	0	1	2	3
0	0	3	6	1
1	5	0	3	6
2	11	10	0	8
3	3	2	5	0

Trace

	0	1	2	3
0	0	3	3	3
1	0	1	2	0
2	3	3	2	3
3	0	1	1	3

□ THUẬT TOÁN DIJKSTRA

Thuật toán Dijkstra

B1: Khởi tạo

- Khởi tạo mảng d , $d[v]$ là độ dài đường đi ngắn nhất từ S tới v . Ban đầu $d[v]$ được khởi gán bằng $c[S, v]$.
- Khởi tạo mảng $Free$, $Free[S]=0$. $Free[v] = 0$ nếu đỉnh v đã xác định được đường đi ngắn nhất (cố định) và ngược lại.
- Khởi tạo mảng $Truoc$, $Truoc[v]=S$ nghĩa là ban đầu mọi đường đi ngắn nhất từ S đến v đều đi thẳng trực tiếp.

B2: Bước lặp (dừng khi không tìm được u)

Chọn trong các đỉnh có nhãn tự do, lấy ra đỉnh u là đỉnh có $d[u]$ nhỏ nhất:

- Cố định nhãn cho u ($Free[u]=0$)
- Xét tất cả các đỉnh tự do v , nếu $d[v] > d[u] + c[u, v]$:
 - $d[v] = d[u] + c[u, v]$
 - $Truoc[v]=u$; (gán lại trên đường đi từ S đến v thì trước v là u)

B3: Sử dụng mảng Truoc truy vết nếu cần.

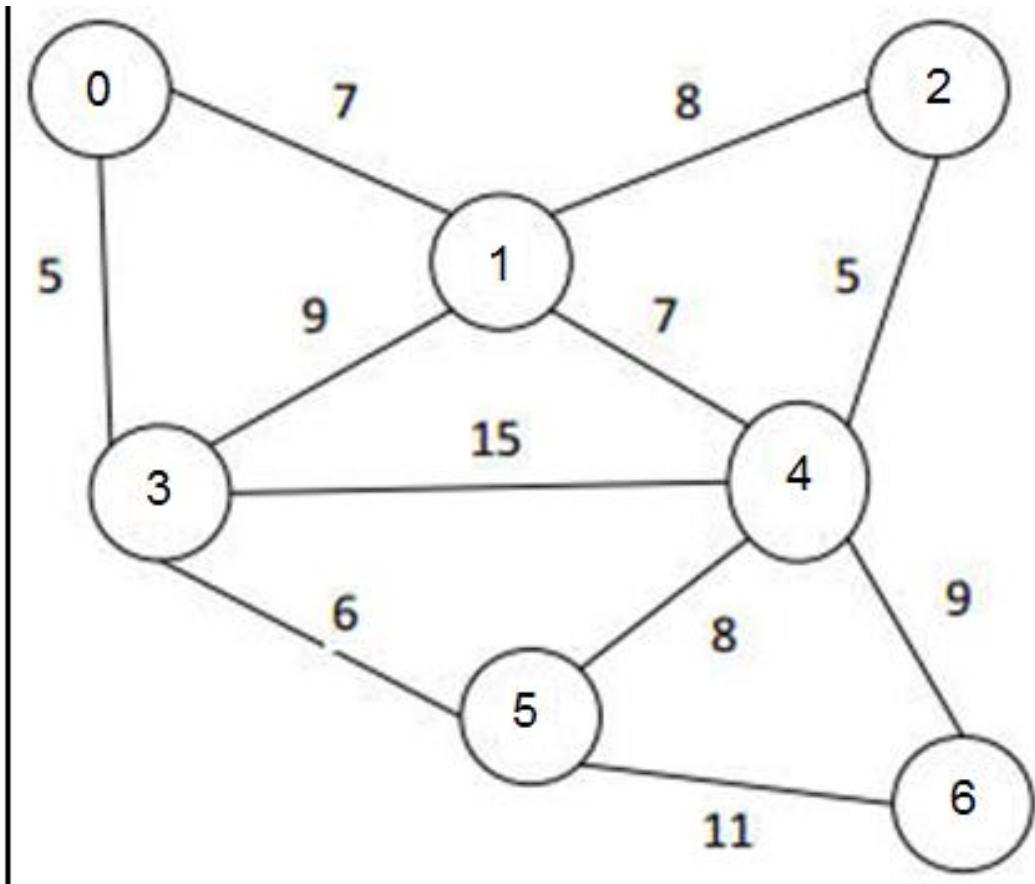
Cây

Bảng băm

Đồ thị

Thuật toán Dijkstra

Đemon thuật toán Dijkstra tìm đường đi ngắn nhất từ đỉnh 0 đến tất cả các đỉnh còn lại.





Ma trận trọng số

Cây

Bảng băm

Đồ thị

Khởi tạo đỉnh xuất phát là $S = 0$

	0	1	2	3	4	5	6
Free	0	1	1	1	1	1	1
d	0	7	999	5	999	999	999
Truoc	0	0	0	0	0	0	0

Bước lặp: Tìm đỉnh u vẫn tự do có $d[u]$ nhỏ nhất

$$\Rightarrow u = 3$$

	0	1	2	3	4	5	6
Free	0	1	1	0	1	1	1
d	0	7	999	5	20	11	999
Truoc	0	0	0	0	3	3	0

Ma trận trọng số

Bước lắp: Tìm đỉnh u vẫn tự do có $d[u]$ nhỏ nhất

$$\Rightarrow u = 1$$

	0	1	2	3	4	5	6
Free	0	0	1	0	1	1	1
d	0	7	15	5	14	11	999
Truoc	0	0	1	0	1	3	0

Bước lắp: Tìm đỉnh u vẫn tự do có $d[u]$ nhỏ nhất

$$\Rightarrow u = 5$$

	0	1	2	3	4	5	6
Free	0	0	1	0	1	0	1
d	0	7	15	5	14	11	22
Truoc	0	0	1	0	1	3	5

Ma trận trọng số

Bước lắp: Tìm đỉnh u vẫn tự do có $d[u]$ nhỏ nhất

$$\Rightarrow u = 4$$

	0	1	2	3	4	5	6
Free	0	0	1	0	0	0	1
d	0	7	15	5	14	11	22
Truoc	0	0	1	0	1	3	5

Bước lắp: Tìm đỉnh u vẫn tự do có $d[u]$ nhỏ nhất

$$\Rightarrow u = 2$$

	0	1	2	3	4	5	6
Free	0	0	0	0	0	0	1
d	0	7	15	5	14	11	22
Truoc	0	0	1	0	1	3	5

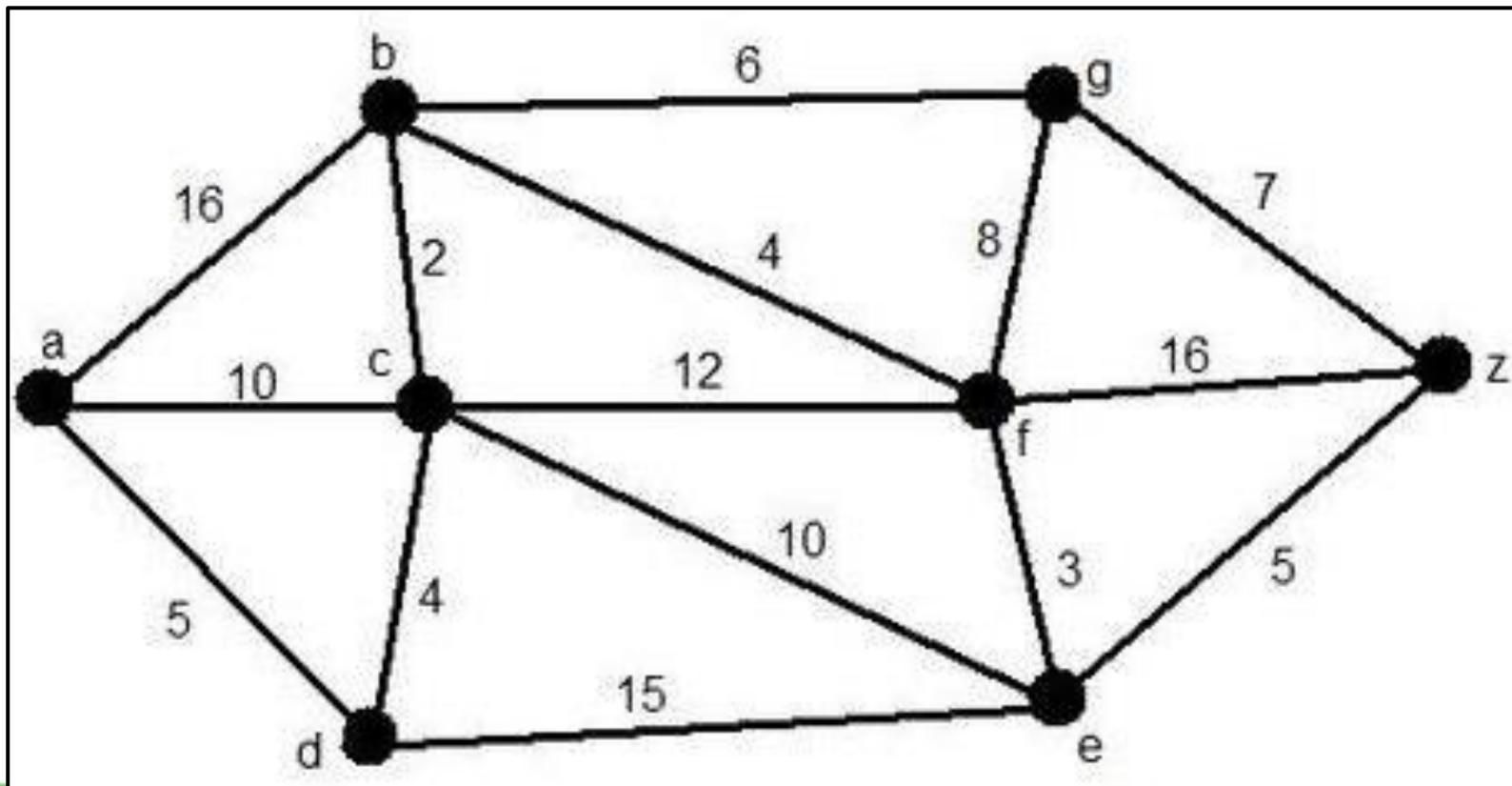
Ma trận trọng số

Bước lắp: Tìm đỉnh u vẫn tự do có $d[u]$ nhỏ nhất
 $\Rightarrow u = 6$

	0	1	2	3	4	5	6
Free	0	0	1	0	0	0	0
d	0	7	15	5	14	11	22
Truoc	0	0	1	0	1	3	5

Thuật toán Dijkstra

Hãy dùng thuật toán Dijkstra tìm đường đi ngắn nhất từ đỉnh a đến tất cả các đỉnh còn lại.



Cây

Bảng băm

Đồ thị



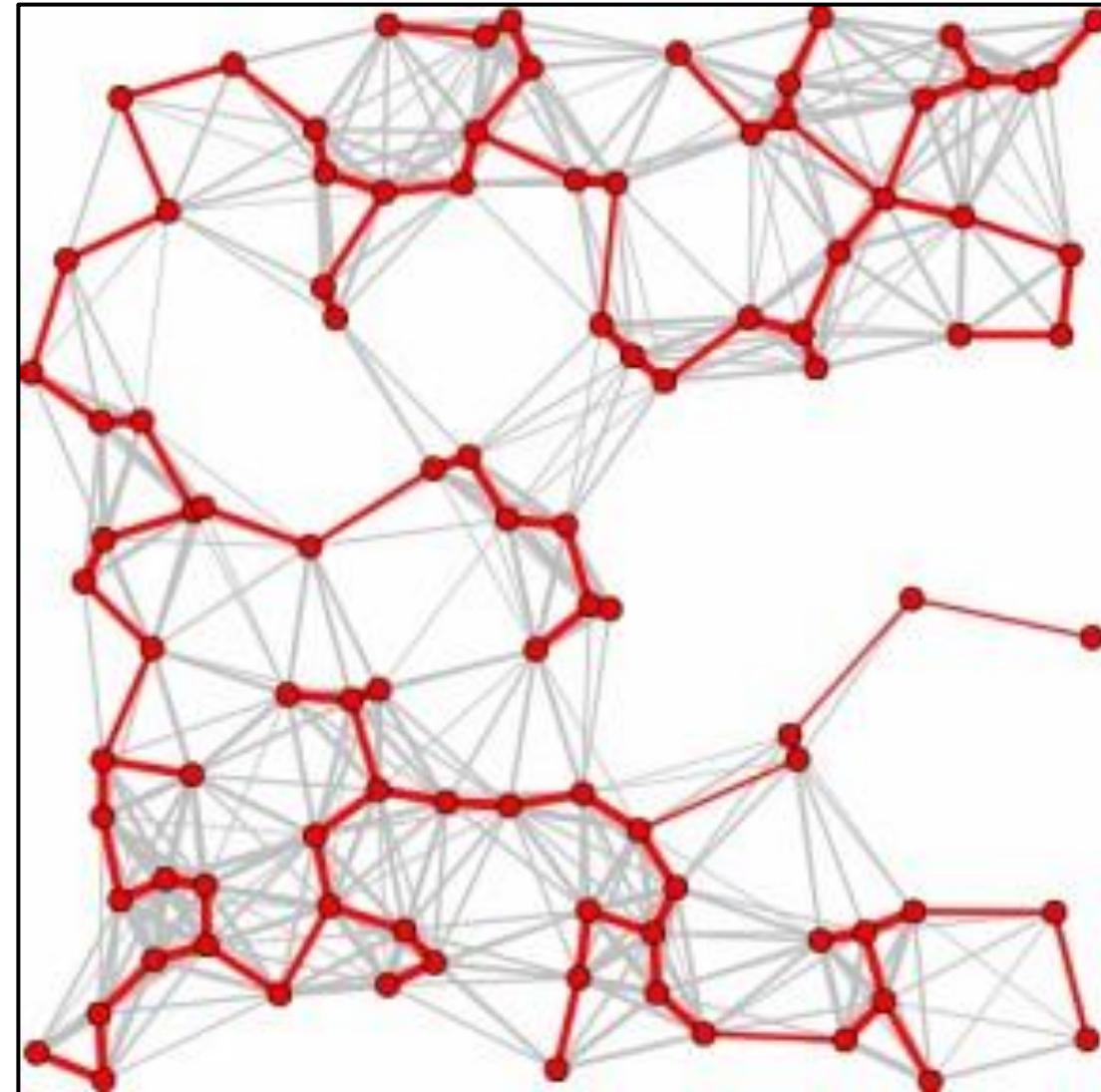
Cây

Bảng băm

Đồ thị

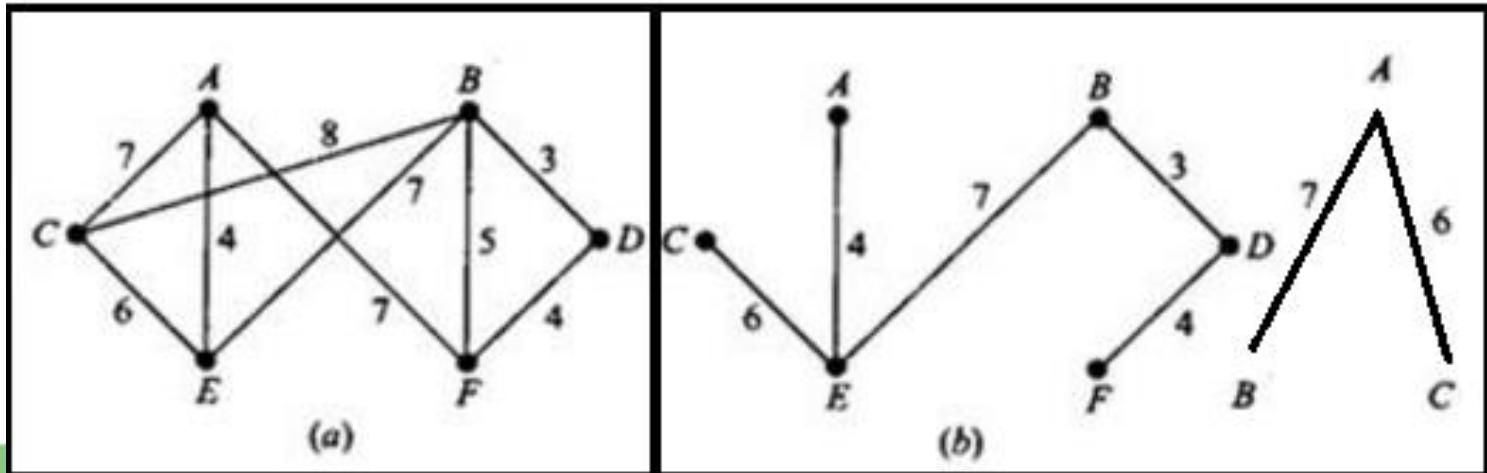
Cây bao trùm ngắn nhất (Kruskal, Prim)

□ CÂY



Định nghĩa

- Cây là một đồ thị mà trong đó hai đỉnh bất kì đều được nối với nhau bằng đúng một đường đi. Nói cách khác một đồ thị vô hướng liên thông không chứa chu trình là một cây.
- Rừng là hợp của các cây. Rừng là đồ thị mà mỗi thành phần liên thông của nó là một cây.



Cây

Bảng băm

Đồ thị

Định lý

Cho đồ thị $G=(V,E)$ có n đỉnh. Sáu mệnh đề sau là tương đương:

1. T là một cây.
2. T liên thông và có $n-1$ cạnh.
3. T không chứa chu trình và có $n-1$ cạnh.
4. T liên thông và mỗi cạnh là cầu
5. Giữa hai đỉnh phân biệt của T luôn có duy nhất một đường đi sơ cấp.
6. T không chứa chu trình nhưng khi bổ sung vào một cạnh nối hai đỉnh không kề nhau thì xuất hiện một chu trình.

Cây

Bảng băm

Đồ thị

CÂY KHUNG(BAO TRÙM) CỦA ĐỒ THỊ

Cây khung của đồ thị

- Mọi đơn đồ thị lên thông G có ít nhất một đồ thị con là cây và chứa tất cả các đỉnh của G . Đồ thị con này được gọi là **cây bao trùm** của G . Đồ thị G có thể có nhiều cây bao trùm.
- Nếu G có trọng số trên các cạnh thì cây bao trùm có tổng trọng số trên các cạnh của nó là nhỏ nhất (lớn nhất) được gọi là cây bao trùm nhỏ nhất (lớn nhất) của đồ thị.
- Trong nội dung chương trình chúng ta sẽ tìm hiểu về 2 thuật toán tìm cây khung nhỏ nhất của đồ thị là: Kruskal và Prim.

Cây

Bảng băm

Đồ thị

□ THUẬT TOÁN KRUSKAL

Thuật toán Kruskal

Với đồ thị vô hướng $G = (V, E)$ có n đỉnh. Khởi tạo cây T ban đầu không có cạnh nào. Xét tất cả các cạnh của đồ thị từ **cạnh có trọng số nhỏ đến cạnh có trọng số lớn**, nếu việc thêm cạnh đó vào T **không tạo thành chu trình** trong T thì **kết nạp thêm cạnh đó** vào T . Cứ làm như vậy cho tới khi:

- Hoặc đã kết nạp được $n-1$ cạnh vào trong T thì ta được T là cây khung nhỏ nhất.
- Hoặc chưa kết nạp đủ $n-1$ cạnh nhưng hễ cứ kết nạp thêm một cạnh bất kỳ trong số các cạnh còn lại thì sẽ tạo thành chu trình đơn. Trong trường hợp này đồ thị G là không liên thông, việc tìm kiếm cây khung thất bại.

Cây

Bảng băm

Đồ thị

Thuật toán Kruskal

Thuật toán có vẻ đơn giản và dễ hiểu, tuy nhiên vẫn đề đặt ra là làm thế nào để kiểm tra việc khi thêm 1 cạnh e vào T thì e sẽ không tạo ra 1 chu trình trong T.

Lưu ý rằng nếu 2 đỉnh u,v thuộc cùng 1 cây T thì khi thêm cạnh (u,v) vào T, T sẽ tồn tại chu trình và ngược lại.

Do đó nếu ta tạo ra một danh sách father, với $\text{father}[i]=$ đỉnh cha của đỉnh i thì việc kiểm tra vẫn đề trên cực kỳ đơn giản và hiệu quả.

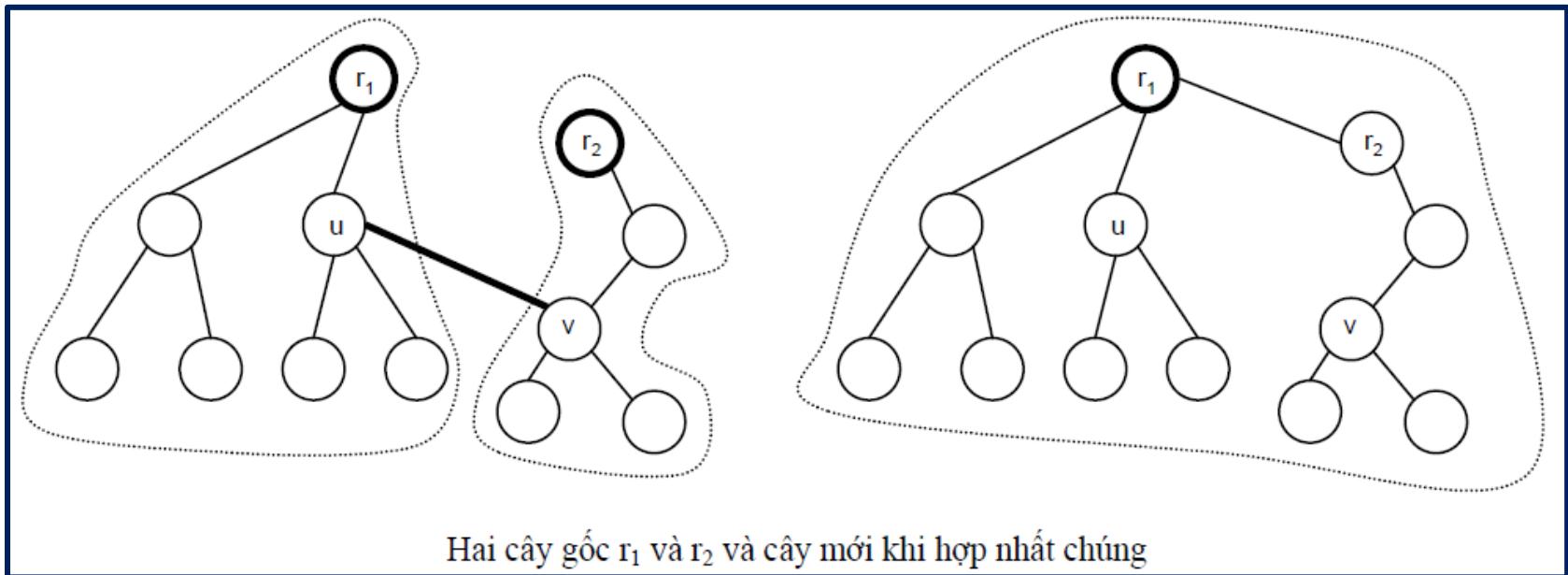
Cây

Bảng băm

Đồ thị

Thuật toán Kruskal

- Gộp 2 cây thành 1 cây



Cây

Bảng băm

Đồ thị

Thuật toán Kruskal

B1: khởi tạo

- Một danh sách các cạnh AE của G đã sắp tăng dần
- Một đồ thị G' có n đỉnh và không có cạnh nào
- Một danh sách *father* có n phần tử , *father*[*i*] = *i* .

B2: Tạo cây khung

- Duyệt qua các cạnh (u,v) của AE từ bé đến lớn. Nếu u,v không cùng gốc và G' ít hơn n-1 cạnh thì
 - Thêm cạnh (u,v) vào G'
 - Nối gốc của u và gốc của v lại với nhau (*father*[*root*(v)]=*root*(u))

Hàm tìm gốc
của đỉnh u

B3: Xuất kết quả

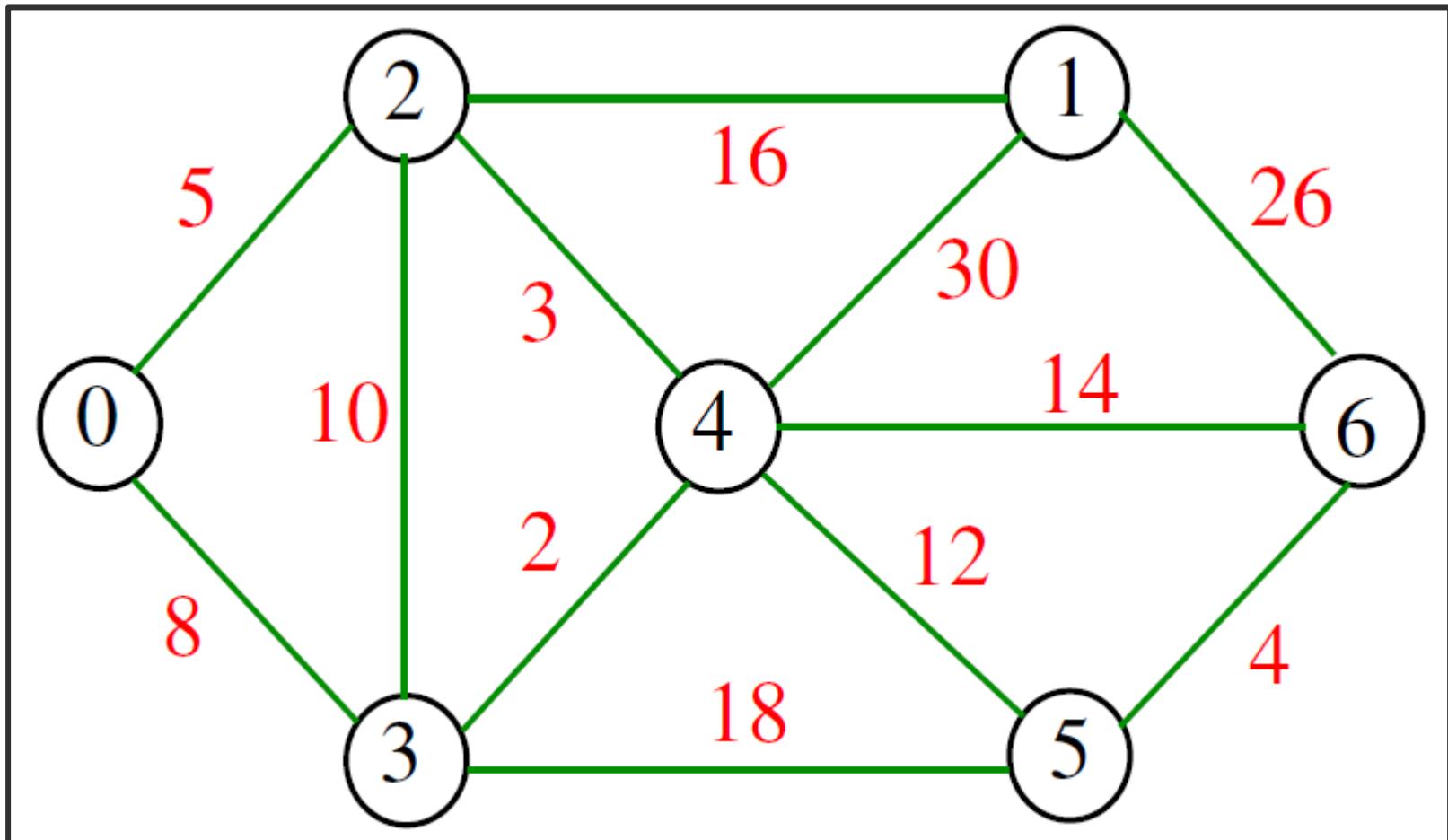
- Nếu G' có n-1 cạnh thì xuất ra cây khung G' của đồ thị G ngược lại G không liên thông

Thuật toán Kruskal

Cây

Bảng băm

Đồ thị



Thuật toán Kruskal

Đồ thị G

DS các
Cạnh

	0	1	2	3	4	5	6
0	0	999	5	8	999	999	999
1	999	0	16	999	30	999	26
2	5	16	0	10	3	999	999
3	8	999	999	0	2	18	999
4	999	30	3	2	0	12	14
5	999	999	999	18	12	0	4
6	999	26	999	999	14	4	0

Đỉnh đầu	0	0	1	1	1	2	2	3	3	4	4	5
Đỉnh cuối	2	3	2	4	6	3	4	4	5	5	6	6
Trọng số	5	8	16	30	26	10	3	2	18	12	14	4

Cây

Bảng băm

Đồ thị

Thuật toán Kruskal

Khởi tạo

Đồ thị G'

DS các Cạnh
đã sắp tăng

Father

	0	1	2	3	4	5	6
0	0	999	999	999	999	999	999
1	999	0	999	999	999	999	999
2	999	999	0	999	999	999	999
3	999	999	999	0	999	999	999
4	999	999	999	999	0	999	999
5	999	999	999	999	999	0	999
6	999	999	999	999	999	999	0

Đỉnh đầu	3	2	5	0	0	2	4	4	1	3	1	1
Đỉnh cuối	4	4	6	2	3	3	5	6	2	5	6	4
Trọng số	2	3	4	5	8	10	12	14	16	18	26	30

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

Thuật toán Kruskal

Xét cạnh 3-4

Đỉnh đầu	3	2	5	0	0	2	4	4	1	3	1	1
Đỉnh cuối	4	4	6	2	3	3	5	6	2	5	6	4
Trọng số	2	3	4	5	8	10	12	14	16	18	26	30

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

Root(3) ≠ root(4) ($3 \neq 4$) →

- thêm cạnh 3-4 vào G'
- Father[root(4)] = root(3)

father[4]=3

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

Cây

Bảng băm

Đồ thị

0	0	999	999	999	999	999	999	999
1	999	0	999	999	999	999	999	999
2	999	999	0	999	999	999	999	999
3	999	999	999	0	2	999	999	999
4	999	999	999	2	0	999	999	999
5	999	999	999	999	999	0	999	999
6	999	999	999	999	999	999	0	999

Đồ thị G'

Thuật toán Kruskal

Xét cạnh 2-4

Đỉnh đầu	3	2	5	0	0	2	4	4	1	3	1	1
Đỉnh cuối	4	4	6	2	3	3	5	6	2	5	6	4
Trọng số	2	3	4	5	8	10	12	14	16	18	26	30

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

Root(2) ≠ root(4) ($2 \neq 3$) →

- thêm cạnh 2-4 vào G'
- Father[root(4)] = root(2)

father[3]=2

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

Cây

Bảng băm

Đồ thị

0	0	999	999	999	999	999	999
1	999	0	999	999	999	999	999
2	999	999	0	999	3	999	999
3	999	999	999	0	2	999	999
4	999	999	3	2	0	999	999
5	999	999	999	999	999	0	999
6	999	999	999	999	999	999	0

Đồ thị G'

Thuật toán Kruskal

Xét cạnh 5-6

Đỉnh đầu	3	2	5	0	0	2	4	4	1	3	1	1
Đỉnh cuối	4	4	6	2	3	3	5	6	2	5	6	4
Trọng số	2	3	4	5	8	10	12	14	16	18	26	30

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

Root(5) ≠ root(6) ($5 \neq 6$) →

- thêm cạnh 5-6 vào G'
- Father[root(6)] = root(5)

father[6]=5

0	1	2	2	3	5	6
0	1	2	2	3	5	5

Cây

Bảng băm

Đồ thị

0	0	999	999	999	999	999	999	999
1	999	0	999	999	999	999	999	999
2	999	999	0	999	3	999	999	999
3	999	999	999	0	2	999	999	999
4	999	999	3	2	0	999	999	999
5	999	999	999	999	999	0	4	
6	999	999	999	999	999	4	0	

Đồ thị G'

Thuật toán Kruskal

Xét cạnh 0-2

Đỉnh đầu	3	2	5	0	0	2	4	4	1	3	1	1
Đỉnh cuối	4	4	6	2	3	3	5	6	2	5	6	4
Trọng số	2	3	4	5	8	10	12	14	16	18	26	30

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

Root(0) ≠ root(2) ($0 \neq 2$) →

- thêm cạnh 0-2 vào G'
- Father[root(2)] = root(0)

father[2]=0

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

Cây

Bảng băm

Đồ thị

0	0	999	5	999	999	999	999
1	999	0	999	999	999	999	999
2	5	999	0	999	3	999	999
3	999	999	999	0	2	999	999
4	999	999	3	2	0	999	999
5	999	999	999	999	999	0	4
6	999	999	999	999	999	4	0

Đồ thị G'

Thuật toán Kruskal

Xét cạnh 0-3

Đỉnh đầu	3	2	5	0	0	2	4	4	1	3	1	1
Đỉnh cuối	4	4	6	2	3	3	5	6	2	5	6	4
Trọng số	2	3	4	5	8	10	12	14	16	18	26	30

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

Root(0) = root(3) ($0 = 0$) →
 – Không thêm cạnh 0-3

Cây

Bảng băm

Đồ thị

0	0	999	5	999	999	999	999
1	999	0	999	999	999	999	999
2	5	999	0	999	3	999	999
3	999	999	999	0	2	999	999
4	999	999	3	2	0	999	999
5	999	999	999	999	999	0	4
6	999	999	999	999	999	4	0

Đồ thị G'

Thuật toán Kruskal

Xét cạnh 2-3

Đỉnh đầu	3	2	5	0	0	2	4	4	1	3	1	1
Đỉnh cuối	4	4	6	2	3	3	5	6	2	5	6	4
Trọng số	2	3	4	5	8	10	12	14	16	18	26	30

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

Root(2) = root(3) ($0 = 0$) →

- Không thêm cạnh 2-3

Cây

Bảng băm

Đồ thị

0	0	999	5	999	999	999	999
1	999	0	999	999	999	999	999
2	5	999	0	999	3	999	999
3	999	999	999	0	2	999	999
4	999	999	3	2	0	999	999
5	999	999	999	999	999	0	4
6	999	999	999	999	999	4	0

Đồ thị G'

Thuật toán Kruskal

Xét cạnh 4-5

Đỉnh đầu	3	2	5	0	0	2	4	4	1	3	1	1
Đỉnh cuối	4	4	6	2	3	3	5	6	2	5	6	4
Trọng số	2	3	4	5	8	10	12	14	16	18	26	30

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

Root(4) ≠ root(5) ($0 \neq 5$) →

- thêm cạnh 4-5 vào G'
- Father[root(5)] = root(4)

father[5]=0

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

Cây

Bảng băm

Đồ thị

0	1	2	3	4	5	6
0	999	5	999	999	999	999
999	0	999	999	999	999	999
5	999	0	999	3	999	999
999	999	999	0	2	999	999
999	999	3	2	0	12	999
999	999	999	999	12	0	4
999	999	999	999	999	4	0

Đồ thị G'

Thuật toán Kruskal

Xét cạnh 4-6

Đỉnh đầu	3	2	5	0	0	2	4	4	1	3	1	1
Đỉnh cuối	4	4	6	2	3	3	5	6	2	5	6	4
Trọng số	2	3	4	5	8	10	12	14	16	18	26	30

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

$$\text{Root}(4) = \text{root}(6) \quad (0 = 0) \rightarrow$$

- Không thêm cạnh 4-6 vào G'

Cây

Bảng băm

Đồ thị

0	0	999	5	999	999	999	999
1	999	0	999	999	999	999	999
2	5	999	0	999	3	999	999
3	999	999	999	0	2	999	999
4	999	999	3	2	0	12	999
5	999	999	999	999	12	0	4
6	999	999	999	999	999	4	0

Đồ thị G'

Thuật toán Kruskal

Xét cạnh 1-2

Đỉnh đầu	3	2	5	0	0	2	4	4	1	3	1	1
Đỉnh cuối	4	4	6	2	3	3	5	6	2	5	6	4
Trọng số	2	3	4	5	8	10	12	14	16	18	26	30

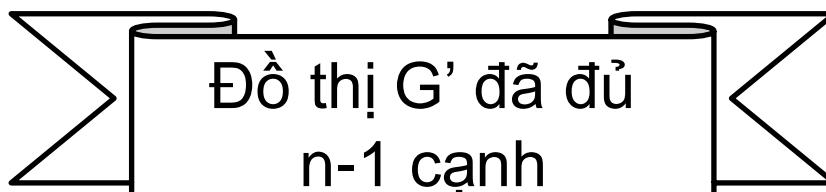
0	1	2	3	4	5	6
0	1	0	2	3	0	5

$\text{Root}(1) \neq \text{root}(2)$ ($1 \neq 0$) \rightarrow

- thêm cạnh 1-2 vào G'
- $\text{Father}[\text{root}(2)] = \text{root}(1)$

father[0]=1

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



Cây

Bảng băm

Đồ thị

0	0	999	5	999	999	999	999
1	999	0	16	999	999	999	999
2	5	16	0	999	3	999	999
3	999	999	999	0	2	999	999
4	999	999	3	2	0	12	999
5	999	999	999	999	12	0	4
6	999	999	999	999	999	4	0

Đồ thị G'

❑ THUẬT TOÁN PRIM

Thuật toán Prim

Thuật toán Kruskal hoạt động chậm trong trường hợp đồ thị dày (có nhiều cạnh). Trong trường hợp đó người ta thường sử dụng phương pháp lân cận gần nhất của Prim. Thuật toán đó có thể phát biểu hình thức như sau:

Cây

Bảng băm

Đồ thị

Thuật toán Prim

Xét cây T trong G và một đỉnh v , gọi **khoảng cách** từ v tới T là trọng số nhỏ nhất trong số các cạnh nối v với một đỉnh nào đó trong T :

$$d[v] = \min\{c[u, v] \mid u \in T\}.$$

Khởi tạo cây T chỉ gồm đỉnh đầu tiên. Sau đó cứ chọn trong số các đỉnh ngoài T ra một đỉnh gần T nhất, kết nạp đỉnh và cạnh có khoảng cách gần nhất đó vào T . Cứ làm như vậy cho tới khi:

- Kết nạp được tất cả n đỉnh $\rightarrow T$ là **cây khung nhỏ nhất**.
- Chưa kết nạp được hết n đỉnh và T không nối với bất kỳ đỉnh nào ngoài $T \rightarrow$ đồ thị không liên thông, tìm cây khung thất bại.

Cây

Bảng băm

Đồ thị

Thuật toán Prim

Về mặt kỹ thuật cài đặt, ta có thể làm như sau:

➤ Sử dụng mảng đánh dấu *Free*.

$Free[v] = 1$ nếu đỉnh v chưa kết nạp vào T

➤ Gọi $d[v]$ là khoảng cách từ v tới T , khởi tạo $d[0] = 0$ và $d[1], d[2], \dots, d[n - 1] = +\infty$.

Tại mỗi bước chọn đỉnh đưa vào T , ta sẽ chọn đỉnh u nào ngoài T và có $d[u]$ nhỏ nhất. Khi kết nạp u vào T rồi thì rõ ràng các khoảng cách $d[v]$ sẽ thay đổi:

$$d[v]_{\text{mới}} = \min(d[v]_{\text{cũ}}, c[u, v]).$$

Cây

Bảng băm

Đồ thị

Thuật toán Prim

B1: Khởi tạo

- Hai mảng đánh dấu $Free[i] = 1$ và $Trace[i] = -1 \forall i$
- Một mảng $d[0] = 0, d[1], d[2], \dots d[n - 1] = +\infty$.

B2: Bước lặp: Trong khi tìm được u có $d[u]$ là nhỏ nhất khác $+\infty$ mà u chưa vẫn còn tự do:

- Đánh dấu $Free[u] = 0$
- Với mọi đỉnh v thỏa $Free[v]=1$ và v kề với u , cập nhật lại $d[v]$ theo công thức:

$$d[v]_{\text{mới}} = \min(d[v]_{\text{cũ}}, c[u, v]) \text{ và } Trace[v] = u \text{ nếu } d[v] \text{ đổi}$$

B3: Xuất kết quả:

- Nếu mọi đỉnh đều đã hết tự do ($Free[i] = 0 \forall i$) thì xuất kết quả trong Trace, Ngược lại đồ thị không liên thông

Cây

Bảng băm

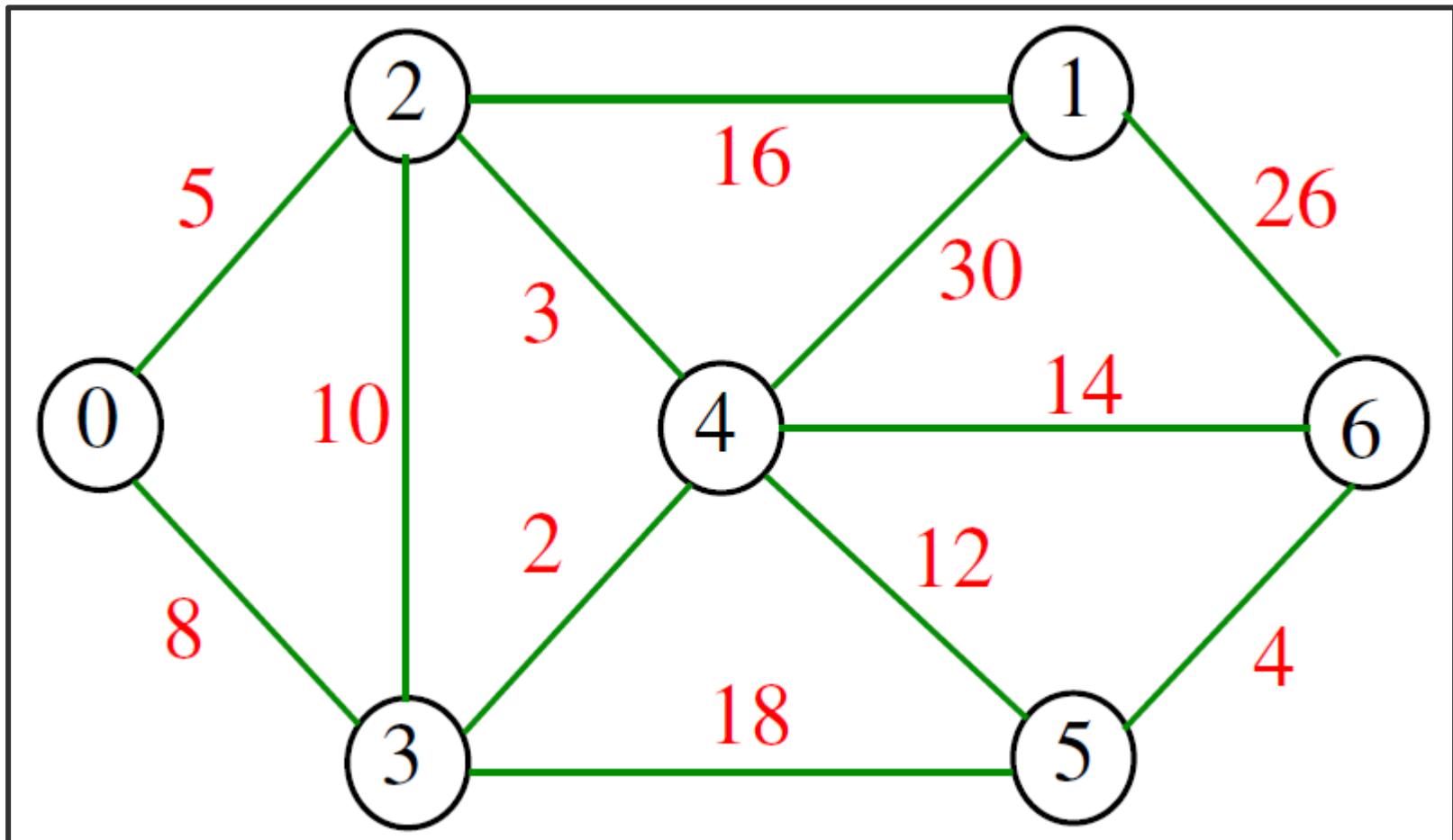
Đồ thị

Thuật toán Prim

Cây

Bảng băm

Đồ thị



Thuật toán Prim

Cây

Bảng băm

Đồ thị

Khởi tạo cây chỉ gồm đỉnh 0 (đỉnh đầu tiên)

	0	1	2	3	4	5	6
Free	1	1	1	1	1	1	1
d	0	999	999	999	999	999	999
Trace	-1	-1	-1	-1	-1	-1	-1

Bước lặp: Tìm đỉnh u vẫn tự do có $d[u]$ nhỏ nhất

$$\Rightarrow u = 0$$

	0	1	2	3	4	5	6
Free	0	1	1	1	1	1	1
d	0	999	5	8	999	999	999
Trace	-1	-1	0	0	-1	-1	-1

Thuật toán Prim

Cây

Bảng băm

Đồ thị

Tìm đỉnh u vẫn tự do có $d[u]$ nhỏ nhất

$$\Rightarrow u = 2$$

	0	1	2	3	4	5	6
Free	0	1	0	1	1	1	1
d	0	16	5	8	3	999	999
Trace	-1	2	0	0	2	-1	-1

Tìm đỉnh u vẫn tự do có $d[u]$ nhỏ nhất

$$\Rightarrow u = 4$$

	0	1	2	3	4	5	6
Free	0	1	0	1	0	1	1
d	0	16	5	2	3	12	14
Trace	-1	2	0	4	2	4	4

Thuật toán Prim

Cây

Bảng băm

Đồ thị

Tìm đỉnh u vẫn tự do có $d[u]$ nhỏ nhất

$$\Rightarrow u = 3$$

	0	1	2	3	4	5	6
Free	0	1	0	0	0	1	1
d	0	16	5	2	3	12	14
Trace	-1	2	0	4	2	4	4

Tìm đỉnh u vẫn tự do có $d[u]$ nhỏ nhất

$$\Rightarrow u = 5$$

	0	1	2	3	4	5	6
Free	0	1	0	0	0	0	1
d	0	16	5	2	3	12	4
Trace	-1	2	0	4	2	4	5

Thuật toán Prim

Cây

Bảng băm

Đồ thị

Tìm đỉnh u vẫn tự do có $d[u]$ nhỏ nhất

$$\Rightarrow u = 6$$

	0	1	2	3	4	5	6
Free	0	1	0	0	0	0	0
d	0	16	5	2	3	12	4
Trace	-1	2	0	4	2	4	5

Tìm đỉnh u vẫn tự do có $d[u]$ nhỏ nhất

$$\Rightarrow u = 1$$

	0	1	2	3	4	5	6
Free	0	0	0	0	0	0	0
d	0	16	5	2	3	12	4
Trace	-1	2	0	4	2	4	5

Tham khảo

- PTS Đinh Mạnh Tường, 2000, Cấu trúc dữ liệu và Thuật toán, Nhà Xuất Bản Khoa Học Và Kỹ Thuật Hà Nội.
- Lê Minh Hoàng, 2011, chuyên đề lý thuyết đồ thị, ĐH Sư phạm Hà Nội.
- Nguyễn Thị Thanh Bình, Nguyễn Văn Phúc, 2010, Cấu trúc dữ liệu và giải thuật 2, ĐH Đà Lạt
- Ngô Quốc Hưng, 2010
[https://sites.google.com/site/ngo2uochung/courses. \(5-2017\)](https://sites.google.com/site/ngo2uochung/courses. (5-2017))

Phụ lục: Bảng mã ASCII

0	20	¶	40	<	60	<	80	P	100	d	120	x	140	î	160	á	180	±	200	£	220	≡	240	
1	21	§	41	>	61	=	81	Q	101	e	121	y	141	ì	161	í	181	±	201	£	221	±	241	
2	22	-	42	*	62	>	82	R	102	f	122	z	142	À	162	ó	182		202	£	222	≥	242	
3	23	±	43	+	63	?	83	S	103	g	123	€	143	ß	163	ú	183		203	£	223	≤	243	
4	24	↑	44	,	64	©	84	T	104	h	124	!	144	É	164	ñ	184	¡	204	£	224	α	244	
5	25	↓	45	-	65	Ã	85	U	105	i	125	›	145	æ	165	Ñ	185	¡	205	=	225	ø	245	
6	26	→	46	.	66	B	86	U	106	j	126	~	146	Œ	166	¤	186		206	£	226	Γ	246	
7	27	←	47	/	67	C	87	W	107	k	127	△	147	ô	167	¤	187]	207	£	227	Π	247	
8	28	↳	48	Ø	68	D	88	X	108	l	128	ç	148	ö	168	ξ	188]	208	£	228	Σ	248	
9	29	↔	49	1	69	E	89	Y	109	m	129	ü	149	ò	169	¬	189	»	209	£	229	σ	249	
10	30	▲	50	2	70	F	90	Z	110	n	130	é	150	û	170	¬	190	±	210	£	230	μ	250	
11	6	31	▼	51	3	71	G	91	[111	o	131	â	151	ù	171	%	191	£	211	τ	231	√	251
12	♀	32	52	4	72	H	92	\	112	p	132	à	152	ü	172	§	192	£	212	£	232	φ	252	
13	33	!	53	5	73	I	93]	113	q	133	à	153	ó	173	!	193	£	213	F	233	θ	253	
14	݂	34	"	54	6	74	J	94	^	114	r	134	ä	154	Ü	174	«	194	£	214	Ω	234	܂	254
15	*	35	#	55	7	75	K	95	–	115	s	135	ç	155	¢	175	»	195	£	215	£	235	δ	255
16	▶	36	\$	56	8	76	L	96	–	116	t	136	é	156	£	176	»	196	–	216	£	236	∞	256
17	◀	37	%	57	9	77	M	97	a	117	u	137	ë	157	¥	177	»	197	+	217	£	237	◊	257
18	‡	38	&	58	:	78	N	98	b	118	v	138	è	158	฿	178	»	198	£	218	£	238	€	258
19	॥	39	,	59	;	79	O	99	c	119	w	139	í	159	ƒ	179	»	199		219	£	239	π	259

Program in Dev-C++

BACK