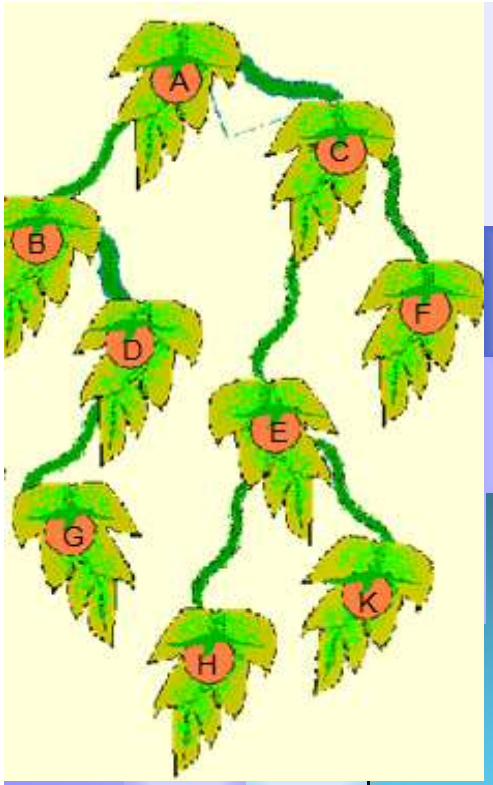


Chương 4: Cây (Tree)

- Nếu ta khảo sát danh sách liên kết mà không quan tâm nhiều đến việc phải duyệt qua tất cả các phần tử, thì ta nên lưu ý một cấu trúc lưu trử phù hợp hơn, đó là cây.
- Cây nhị phân tìm kiếm là một cấu trúc dữ liệu đặc biệt được sử dụng cho mục đích lưu trữ dữ liệu, nhưng nó tận dụng được lợi thế của hai cấu trúc dữ liệu:
  - Mảng đã có thứ tự : việc tìm kiếm sẽ nhanh như trong mảng đã sắp thứ tự
  - Danh sách liên kết đơn: các thao tác chèn và xóa cũng nhanh như trong danh sách liên kết đơn.
- . . .

Mục tiêu

Tiếp cận các cấu trúc dữ liệu và các phép toán đặc trưng trên câu trúc dữ liệu tương ứng :

- Cây nhị phân.
- Cây nhị phân tìm kiếm.
- Cây cân bằng.

Nội dung :

- Cấu trúc cây
- Cây nhị phân
- Cây nhị phân tìm kiếm
- Cây cân bằng

Cấu trúc cây

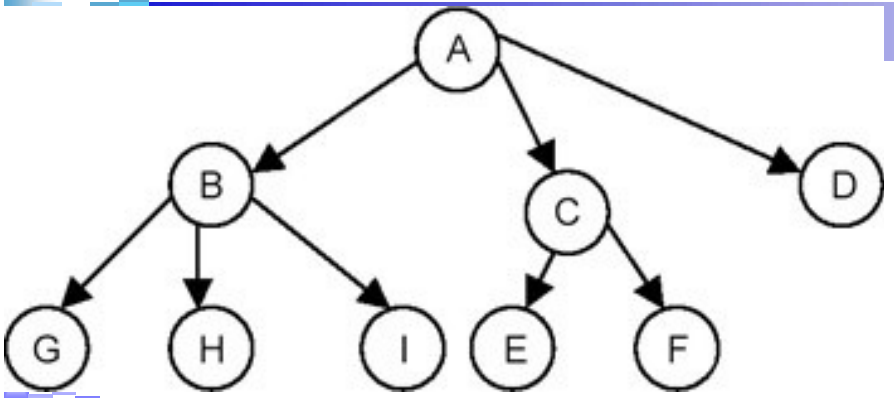
Định nghĩa:

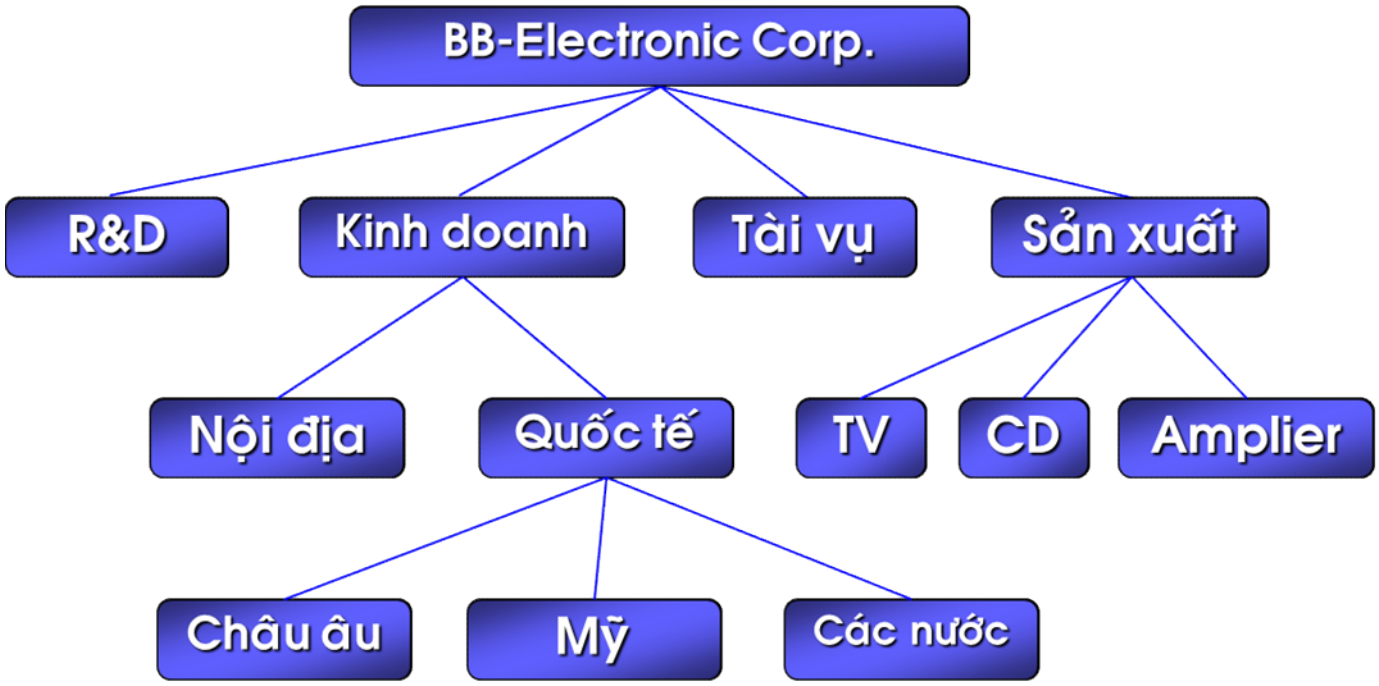
- Cây là một tập hợp các phần tử gọi là nút (nodes) trong đó có một nút đặc biệt được gọi là nút gốc (root).
- Trên tập hợp các nút này có một quan hệ, gọi là quan hệ cha - con (parenthood), để xác định hệ thống cấu trúc trên các nút. Mỗi nút, trừ nút gốc, có duy nhất một nút cha. Một nút có thể có một hay nhiều nút con hoặc không có nút con nào. Nút gốc không có nút cha.
- Quan hệ cha con được biểu diễn theo qui ước nút cha ở dòng trên nút con ở dòng dưới và được nối bởi một đoạn thẳng.
- Mỗi nút có thể có kiểu nào đó bất kỳ.

Ta có thể định nghĩa cây dưới dạng đệ quy như sau:

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

  Cây mới này có nút gốc là n và các cây T_1,.., T_k gọi là các cây con.
- Tập rỗng cũng được coi là một cây và gọi là cây rỗng kí hiệu ϕ (khi cài đặt là NULL)





Sơ đồ tổ chức của một công ty

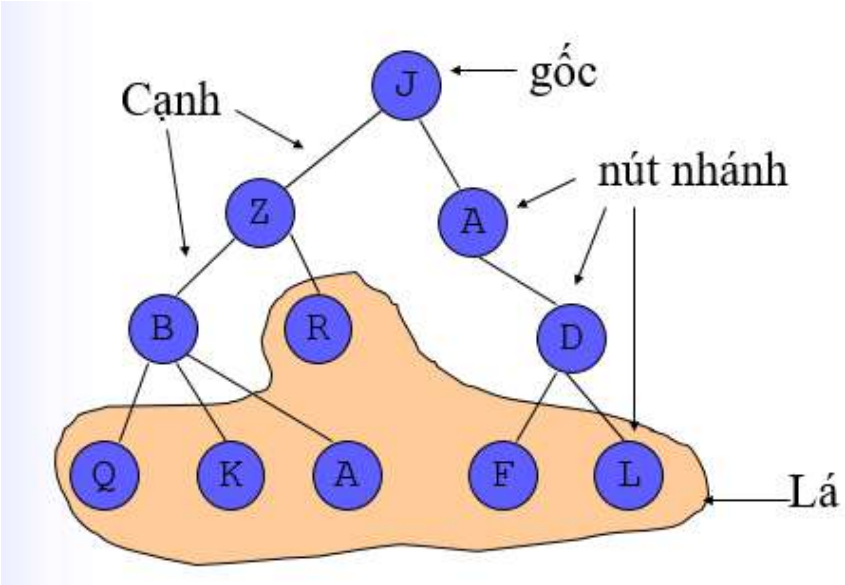
Một số khái niệm cơ bản

Nút gốc : là nút không có nút cha.

Nút lá : là nút không có con.

Nút nhánh : là nút không phải nút lá và không phải là gốc .

Cạnh (nhánh): Đoạn nối từ nút x đến con y của nó gọi là cạnh (nhánh) từ x đến y

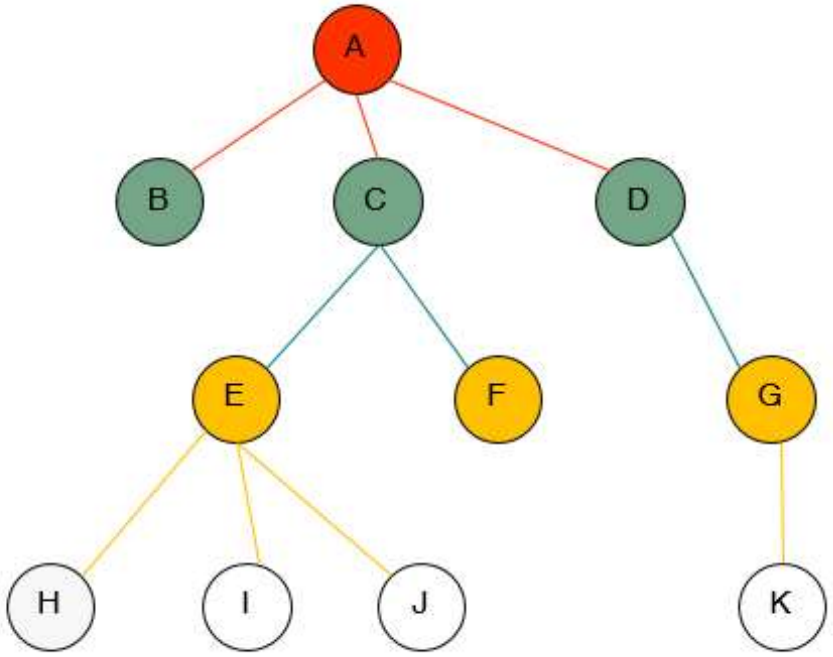


- **Bậc của một nút :** là số cây con của nút đó, (Nút lá có bậc 0)

  Bậc của A : 3; Bậc của D : 1; Bậc của C : 2; Bậc của K : 0**;**
- **Bậc của một cây :** là bậc lớn nhất của các nút trong cây (số cây con lớn nhất của các nút thuộc cây).

  Cây có bậc n thì gọi là cây n-phân.

(cây 3-phân)



- **Đường đi từ x đến y** : Đường đi từ nút x đến nút y là chuỗi các nút $(x,x_1,..., x_n,y)$, trong đó x là cha của x_1, x_n là cha của y, x_i là cha của x_{i+1}, i = 1,.., n-1.

  Đường đi từ A đến K : (A,D,G,K)

  Đường đi từ A đến F : (A,C,F)

  Lưu ý : Từ gốc đến nút x có và chỉ có m đường đi.

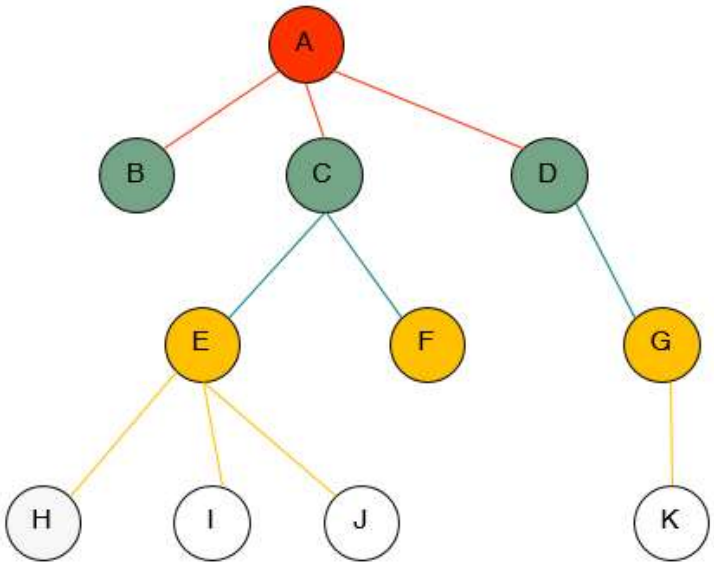
- **Độ dài đường đi từ gốc đến nút x :** là cạnh cần đi qua kể từ gốc đến x

  Độ dài đường đi từ A đến K : 3 (3 cạnh AD,DG,GK)

  Độ dài đường đi từ A đến F : 2

- **Chiều cao của cây :** độ dài đường đi lớn nhất từ gốc đến các nút lá .

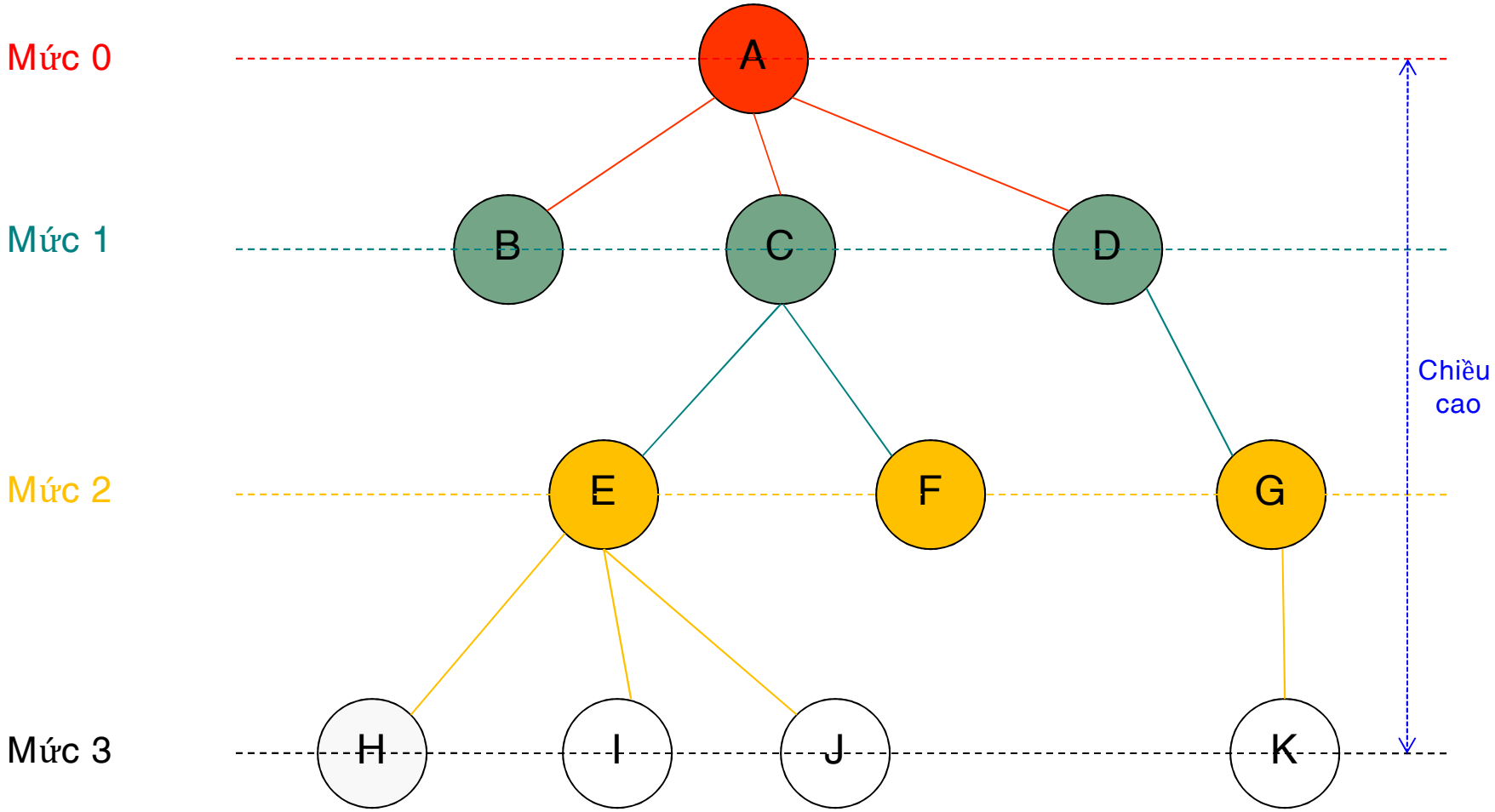
- **Rừng cây:** là tập hợp nhiều cây rời nhau.

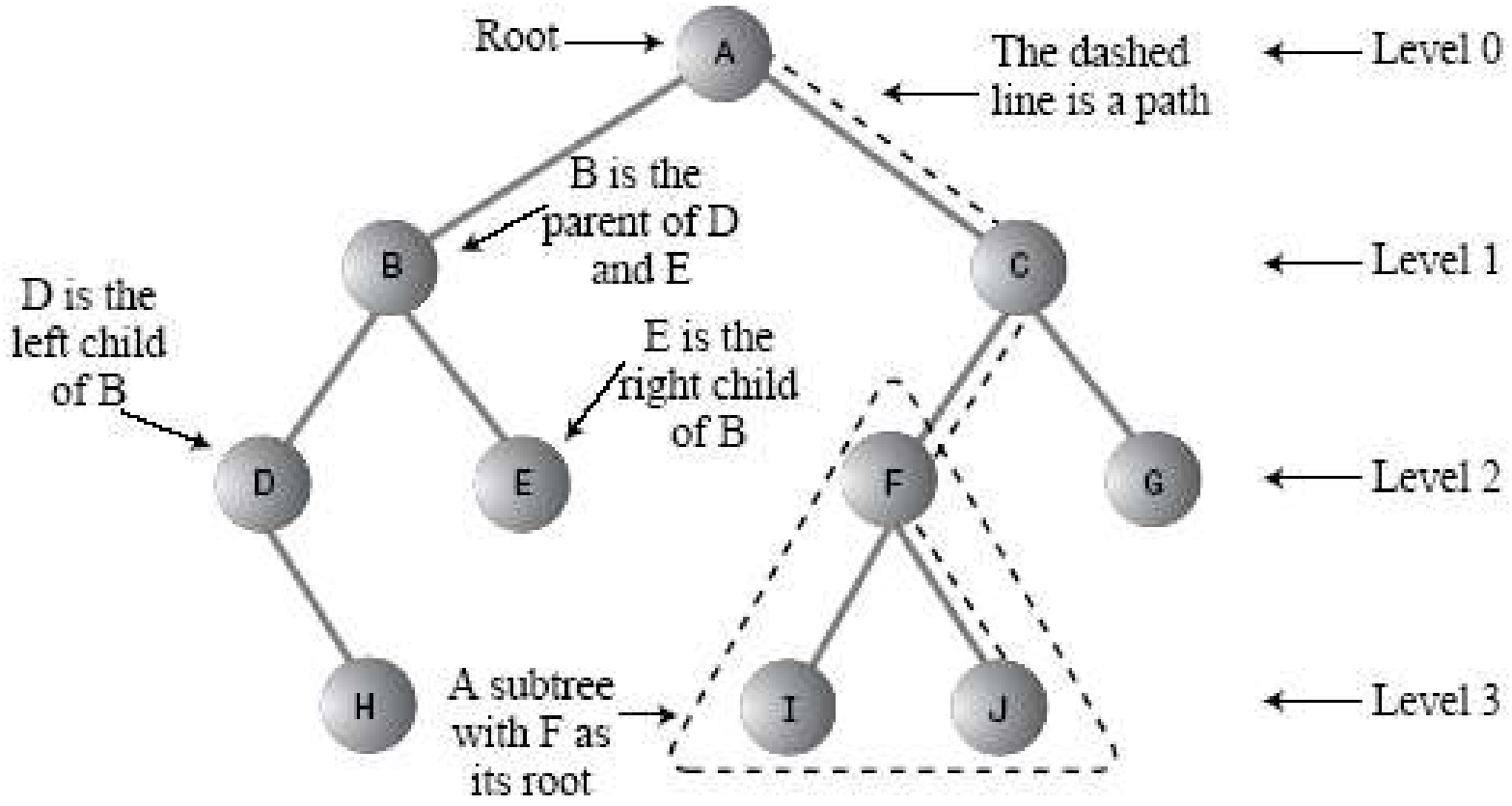


- **Mức của một nút :**
  - Mức (gốc (T)) = 0.
  - Gọi $T_1, T_2, T_3, \ldots, T_n$ là các con của T_0 :

    Mức (gốc T_1) = Mức (gốc T_2) = ... = Mức (gốc Tn) = Mức (gốc T_0) + 1.

- *Nhận xét:*
  - *Mức của nút x = số cạnh từ gốc đến x*
  - *Chiều cao của cây là mức lớn nhất của các nút lá*


Mức 0
Mức 1
Mức 2
Mức 3
A
B
C
D
E
F
G
H
I
J
K
Chiều cao


Root
A
The dashed
line is a path
Level 0
B is the
parent of D
and E
B
C
Level 1
D is the
left child
of B
E is the
right child
of B
D
E
F
G
Level 2
H
A subtree
with F as
its root
I
J
Level 3

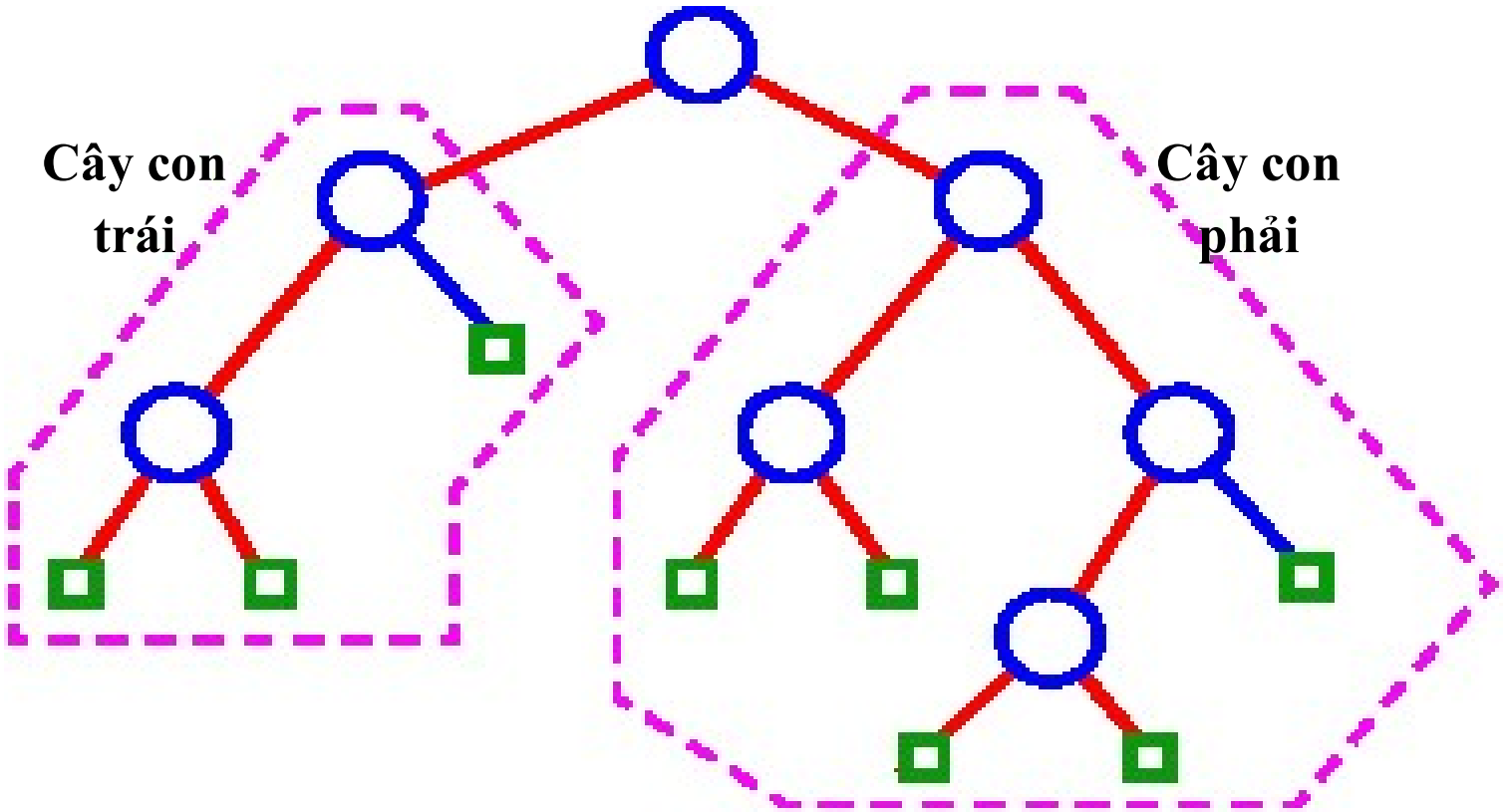
Nhận xét:

- Cây không tồn tại chu trình.

Cây nhị phân
(Binary Trees)

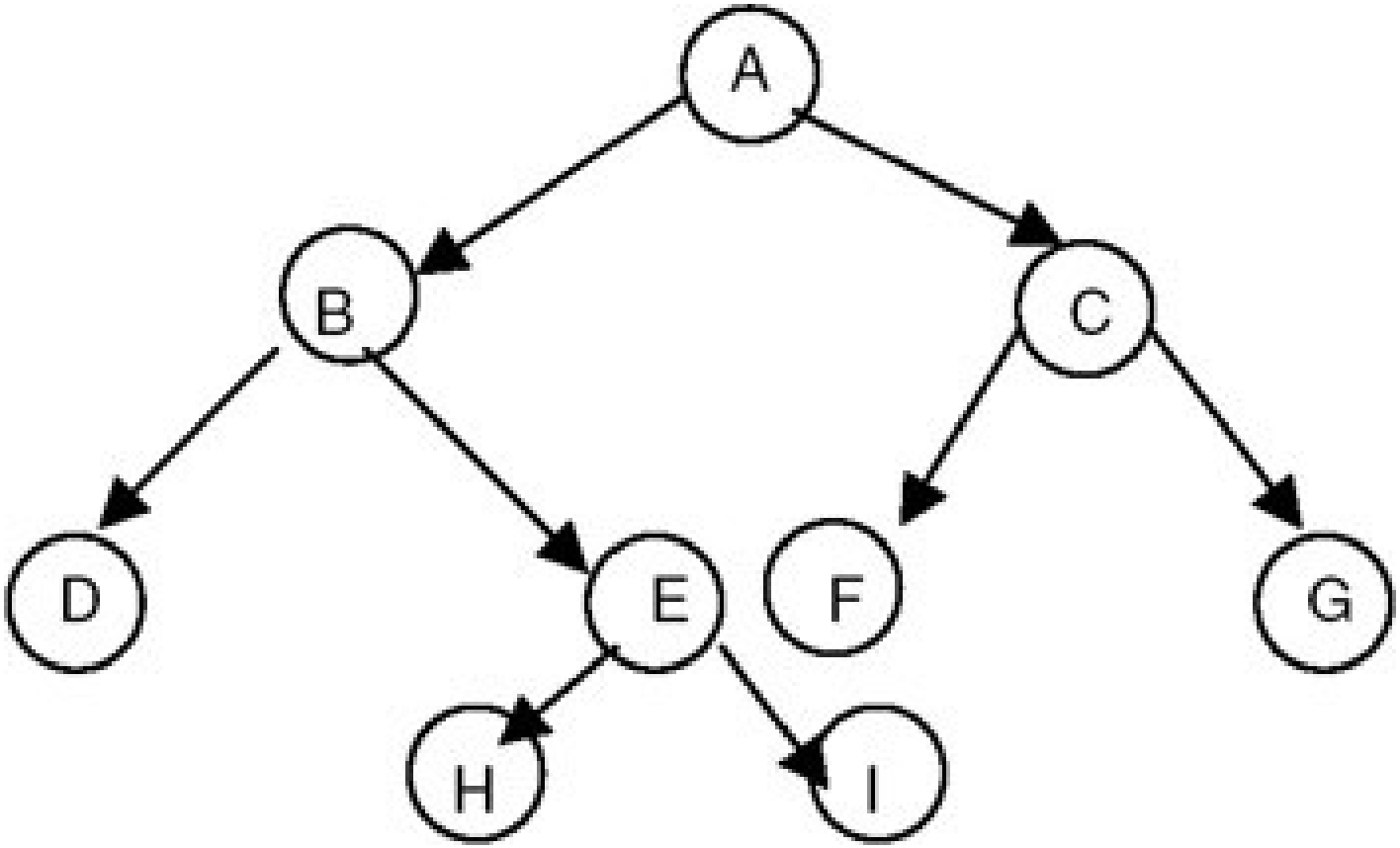
- **Định nghĩa:** Cây nhị phân là cây mà mỗi nút có tối đa 2 cây con

Cây nhị phân



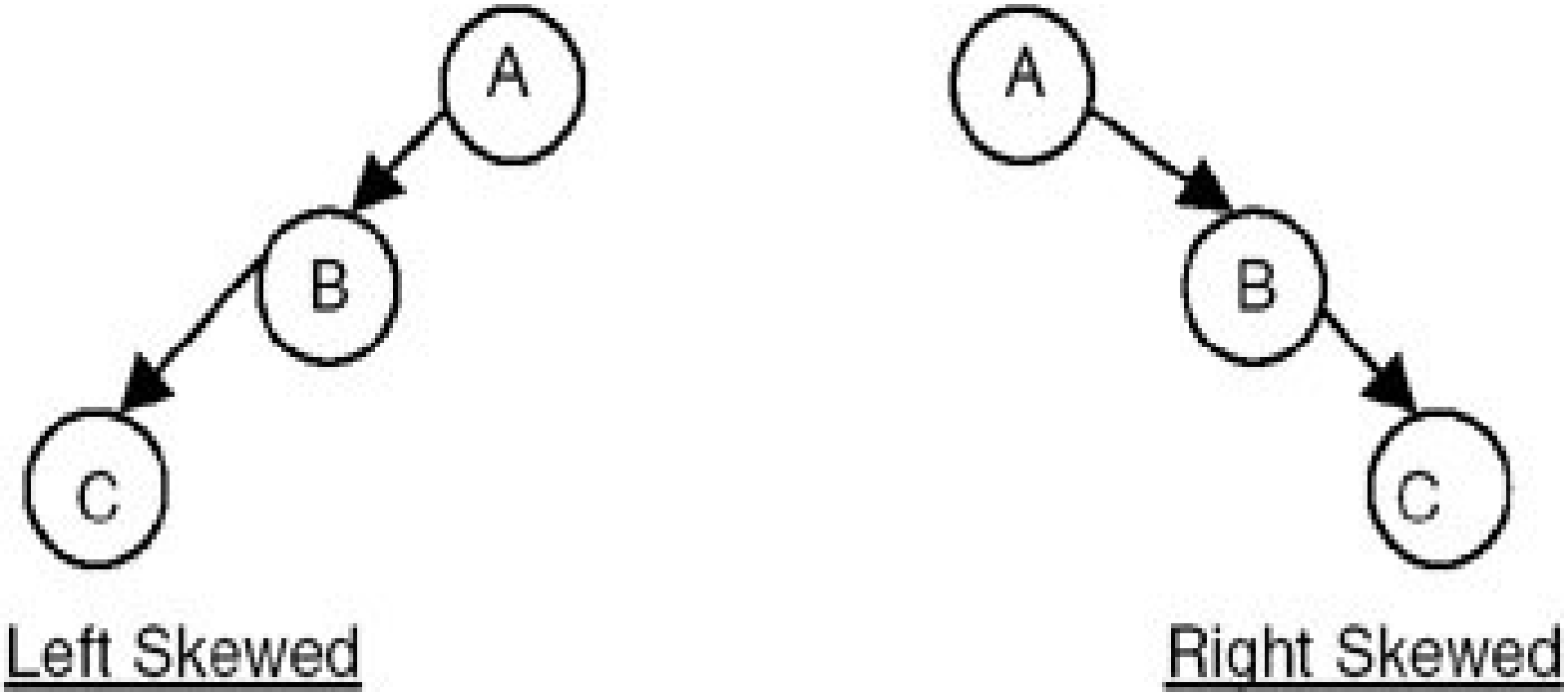
Hình ảnh một cây nhị phân

Cây nhị phân



Binary tree structure.

Cây nhị phân



Skewed trees.

Cây nhị phân

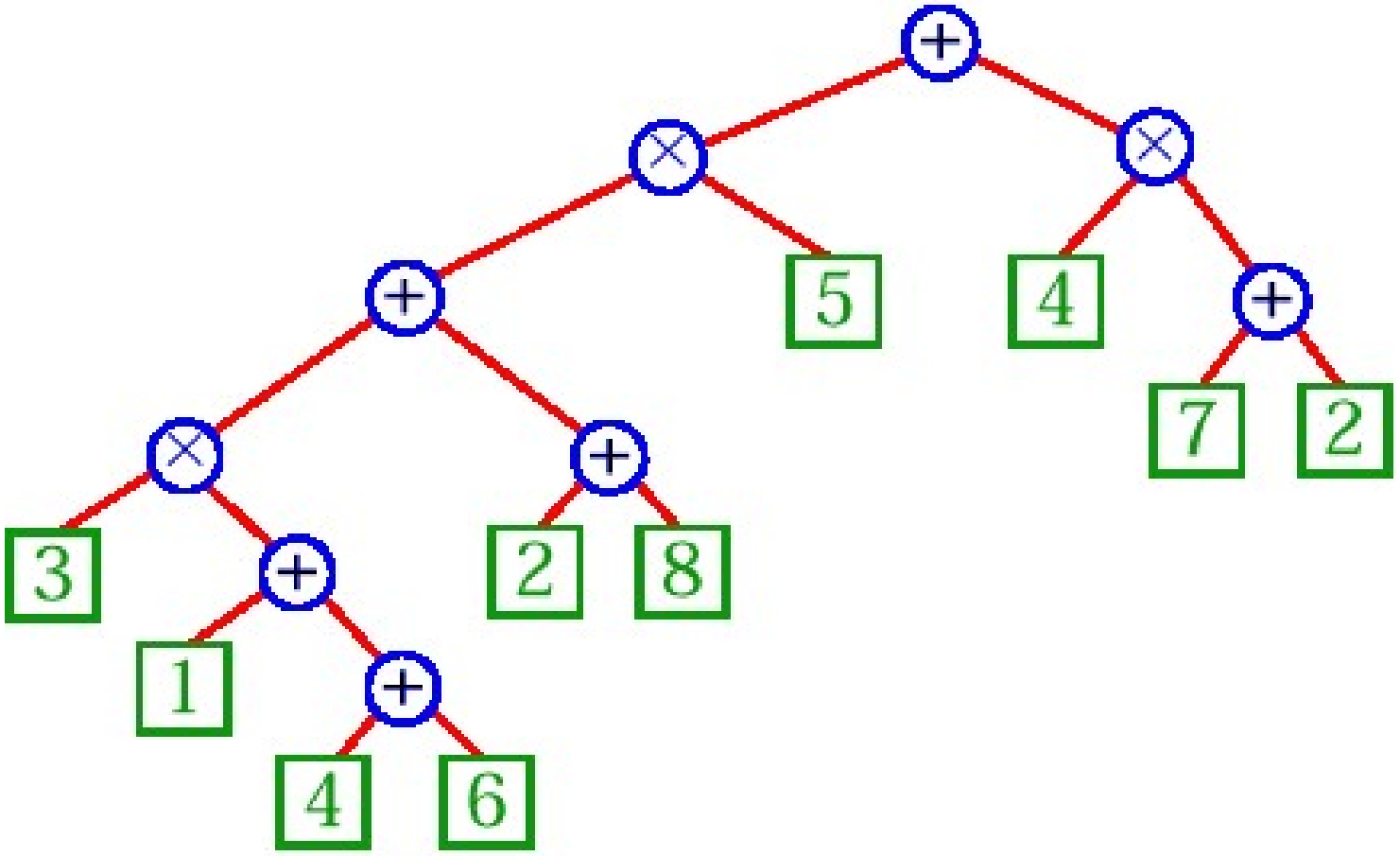
- Cây nhị phân dùng để biểu diễn một biểu thức toán học:

$$((((3 \times (1 + (4 + 6))) + (2 + 8)) \times 5) + (4 \times (7 + 2)))$$

Cây nhị phân

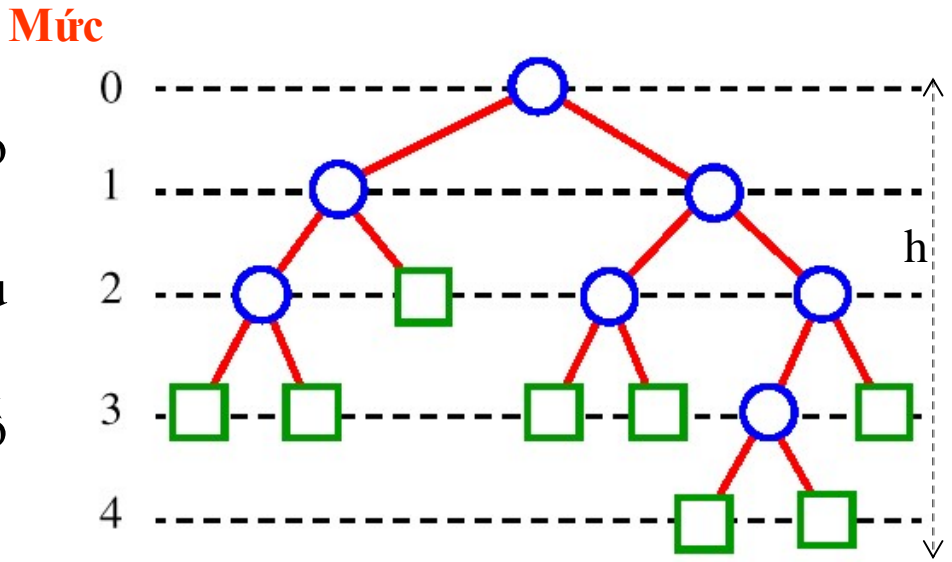
- Cây nhị phân dùng để biểu diễn một biểu thức toán học:

$((((3 \times (1 + (4 + 6))) + (2 + 8)) \times 5) + (4 \times (7 + 2)))$



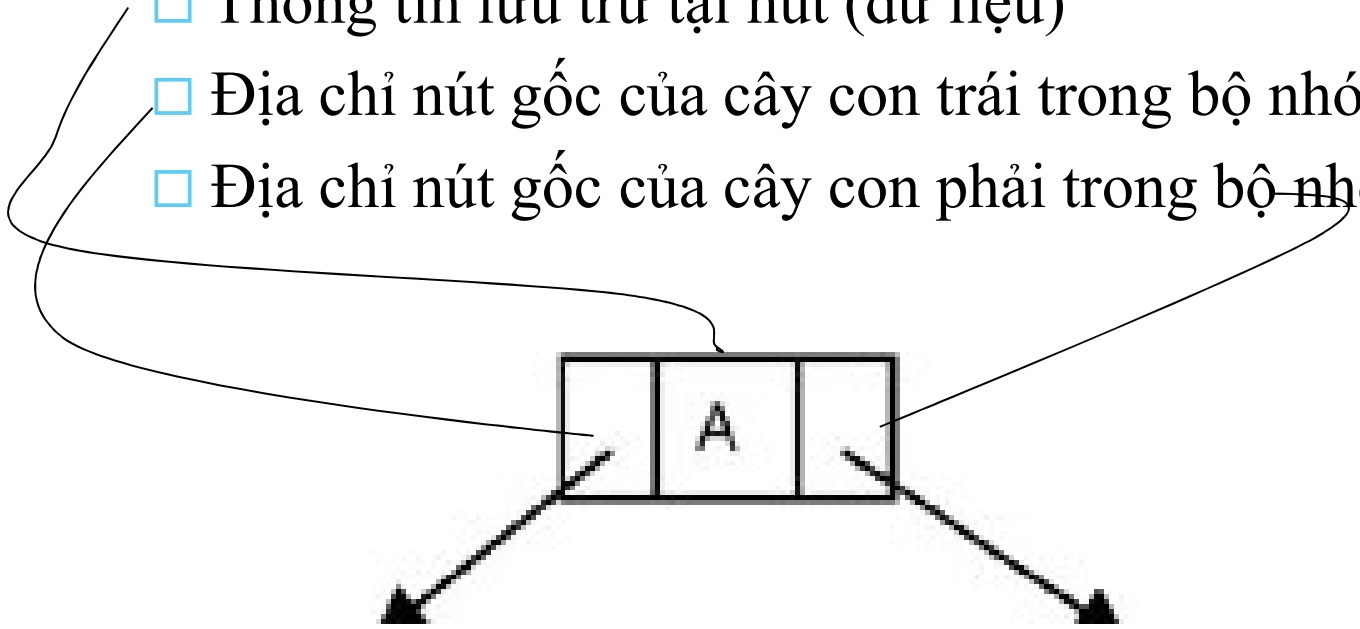
Một số tính chất của cây nhị phân

- Số nút nằm ở mức i $\leq 2^i$
- Chiều cao cây h là mức cao nhất của các nút (lá).
- Số nút lá $\leq 2^h$, với h là chiều cao của cây.
- Chiều cao của cây h $\geq \log_2$(số nút trong cây).
- Số nút trong cây $\leq 2^{h+1}-1$.

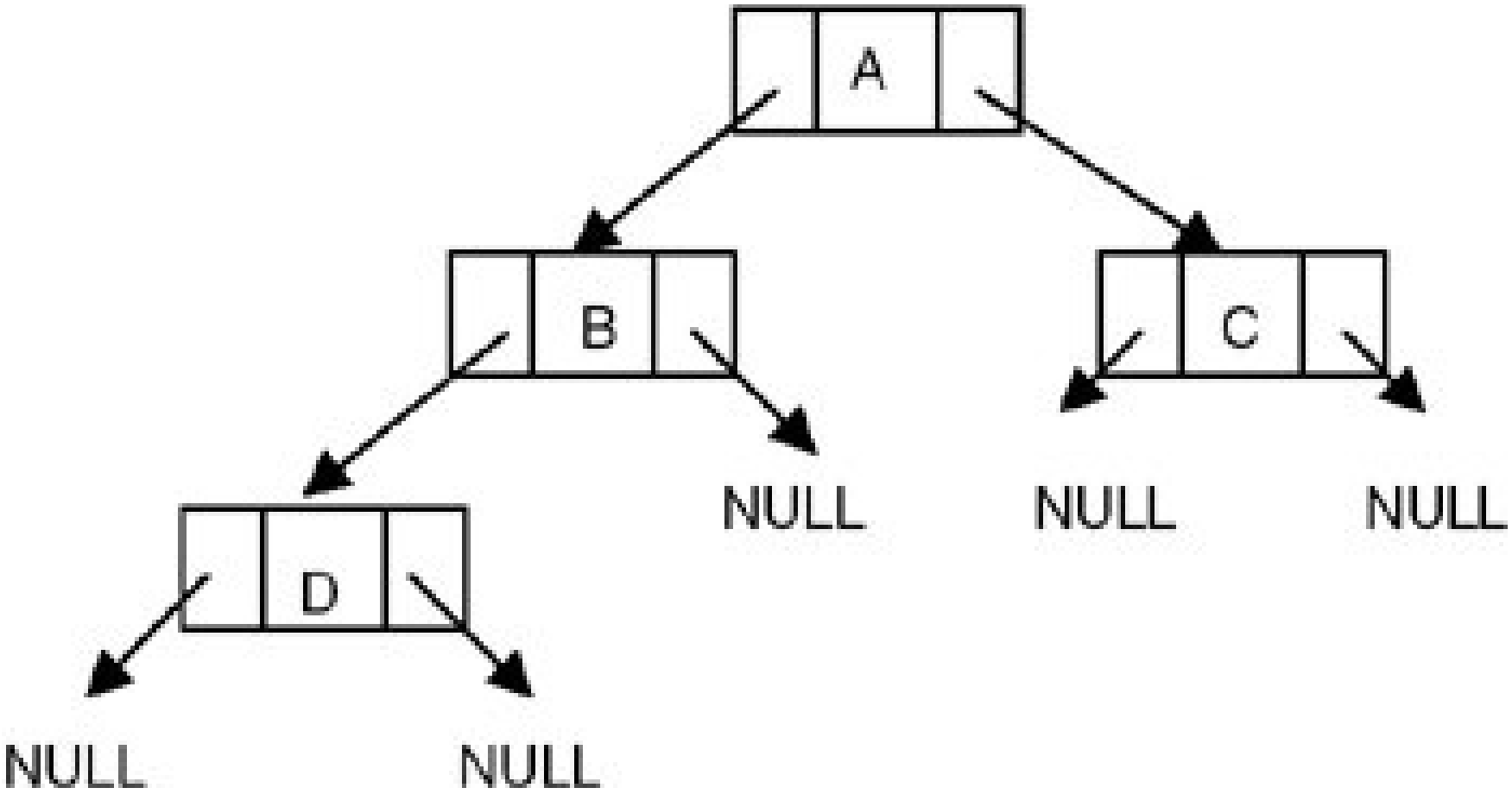


Biểu diễn cây nhị phân T

- Cây nhị phân là một cấu trúc bao gồm các phần tử (nút) được kết nối với nhau theo quan hệ "cha-con" với mỗi cha có tối đa 2 con. Để biểu diễn cây nhị phân ta chọn phương pháp cấp phát liên kết. Ứng với một nút, ta sử dụng một biến động lưu trữ các thông tin sau:
  - Thông tin lưu trữ tại nút (dữ liệu)
  - Địa chỉ nút gốc của cây con trái trong bộ nhớ.
  - Địa chỉ nút gốc của cây con phải trong bộ nhớ.



Cây nhị phân

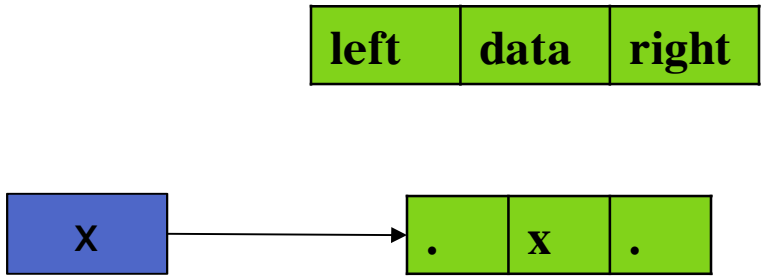


Cài đặt cấu trúc dữ liệu cây nhị phân

```
typedef  < kiểu dữ liệu của thành phần dữ liệu các nút >    TData;
//Kiểu các nút trong cây
struct TNode
{
        TData data;
        TNode* left;
         TNode* right;
};
//Kiểu cây nhị phân : Kiểu con trỏ trỏ đến các phần tử kiểu TNode
typedef  TNode * Tree;
```

1. Tạo nút với dữ liệu x cho trước: CreateTNode

```
struct TNode
{
        TData data;
        TNode* left;
        TNode* right;
};
//Kiểu cây nhị phân : Kiểu con trỏ trỏ đến các
phần tử kiểu TNode
typedef  TNode * Tree;
```



2. Tạo cây rỗng: CreateTree

3. Kiểm tra nút lá

Một số thao tác cơ bản trên cây nhị phân :

```
//1. Tạo nút với dữ liệu x cho trước : CreateTNode
//Input x
//Output : NULL ; khong thanh cong
//          p; tro den nut vua tao, neu thanh cong
TNode  *CreateTNode(TData x) // ≡ GetNode : tao nut
{
        TNode *p = new TNode;
        if (p != NULL)
        {
                p->data = x;
                p->left = NULL;
                p->right = NULL;
        }
        return p;
}
```

2. **Tạo cây rỗng**

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

```
void CreateTree(Tree *T)
{
        (*T)=NULL;
}
```

3. **Kiểm tra cây rỗng**

```
int EmptyTree(Tree T)
{
        if (T == NULL)
                return 1;
        return 0;
}
```

4. Hàm trả về con trái của một nút

```
Tree LeftChild(Tree T)
{
        if (T!=NULL)
            return T->left;
        return NULL;
}
```

//Nếu **LeftChild(Tree T) == NULL** thì do T == NULL hoặc T->left == NULL

5. Hàm trả về con phải của một nút

```
Tree RightChild(Tree T)
{
        if (T!=NULL)
            return T->right;
        return NULL;
}
```

//Nếu **RightChild(Tree T) == NULL** thì do T == NULL hoặc T->right == NULL

6. Kiểm tra nút lá:

- Một nút là nút lá thì nó không có một con nào cả, nên khi đó con trái và con phải của nó cùng bằng NULL

```
int IsLeaf(Tree T)
{
        int kq = 0;
        if(T!=NULL)
                kq = (LeftChild(T)==NULL)&&(RightChild(T)==NULL);
        return kq;
}
```

7. Xác định số nút của cây

```
int CN_Nodes(Tree T)
{
        if(EmptyTree(T))
                return 0;
        return 1+CN_Nodes(LeftChild(T)) + CN_Nodes(RightChild(T));
}
```

8. Duyệt cây nhị phân

- Có 3 kiểu duyệt chính có thể áp dụng trên cây nhị phân:
  - Duyệt theo thứ tự trước (tiền tự)- **Preorder**
  - Duyệt theo thứ tự giữa (trung tự)- **Inorder**
  - Duyệt theo thứ tự sau (hậu tự)- **Postorder**
- Tên của 3 kiểu duyệt này được đặt dựa trên trình tự của việc thăm nút gốc so với việc thăm 2 cây con.

Duyệt theo thứ tự trước (PreOrder):

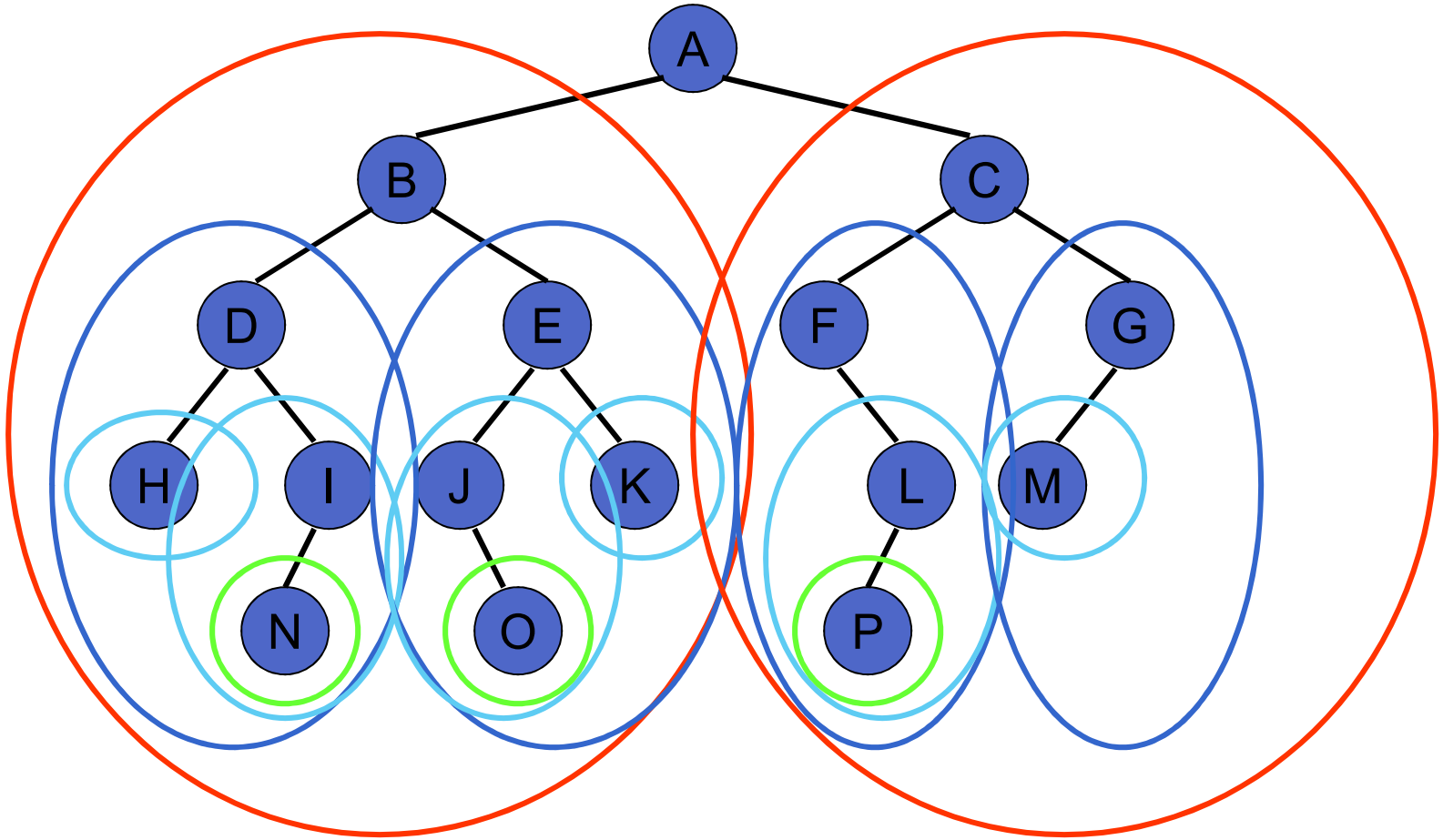
Node – Left – Right

NLR

Kiểu duyệt này : Trước tiên thăm nút gốc sau đó thăm các nút của cây con trái rồi đến cây con phải (NLR)

```
void PreOrder(Tree T)
{
        if( T != NULL)
        {
            <Xử lý nút>
            PreOrder(T->left);
            PreOrder(T->right);
        }
}
```

Duyệt theo thứ tự trước (**PreOrder) :** Node-Left-Right



Kết quả: A B D H I N E J O K C F L P G M

Duyệt theo thứ tự giữa (Inorder)

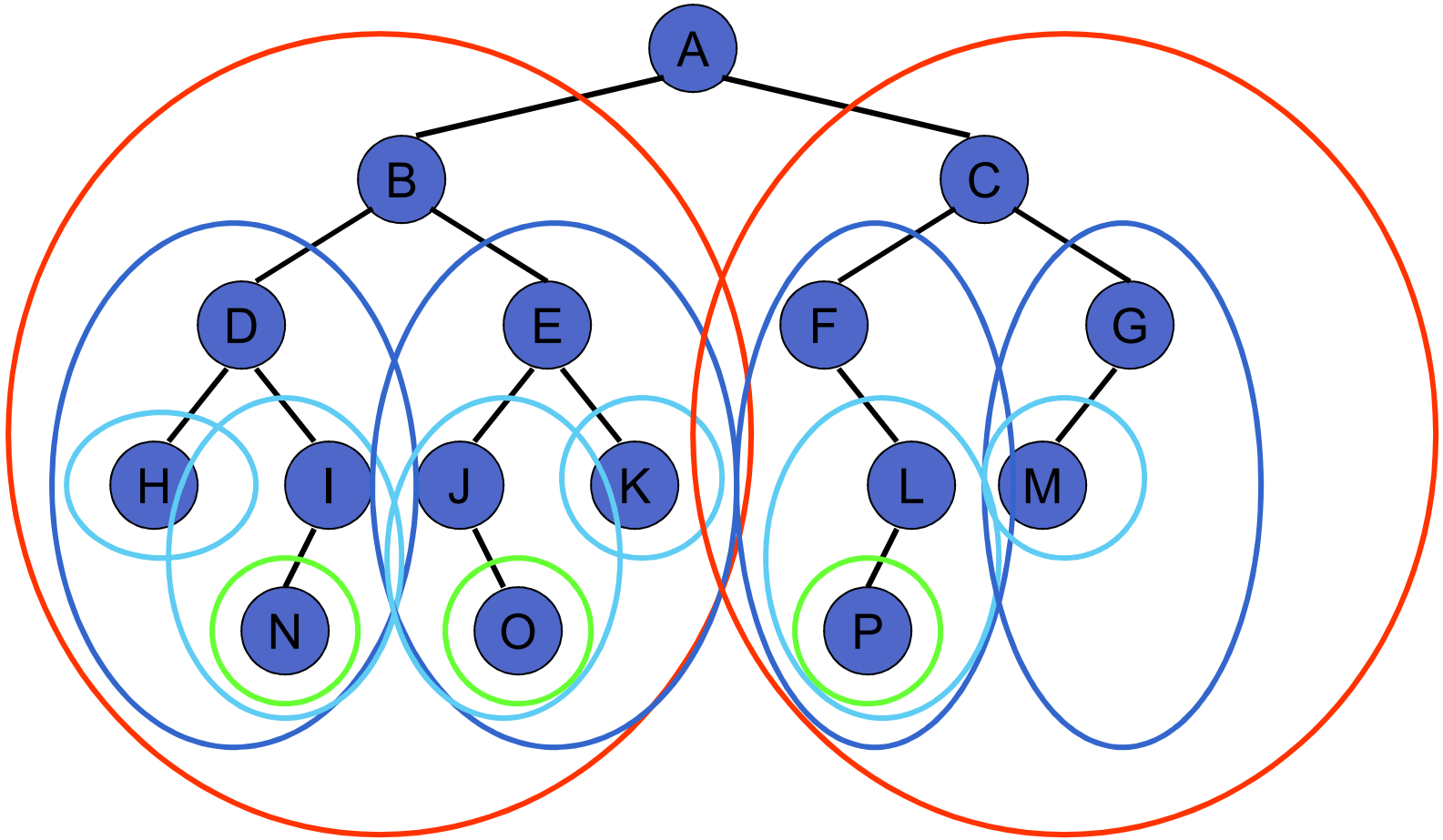
Left - Node –Right

LNR

- Kiểu duyệt này trước tiên thăm các nút của cây con trái sau đó thăm nút gốc rồi đến cây con phải (LNR)

```
void InOrder(Tree T)
{
        if( T != NULL)
        {
                InOrder(T->left);
                <Xử lý nút>
                InOrder(T->right);
        }
}
```

Duyệt theo thứ tự giữa (Left- Node-Right)



Kết quả: H D N I B J O E K A F P L C M G

Duyệt theo thứ tự sau (PosOrder)

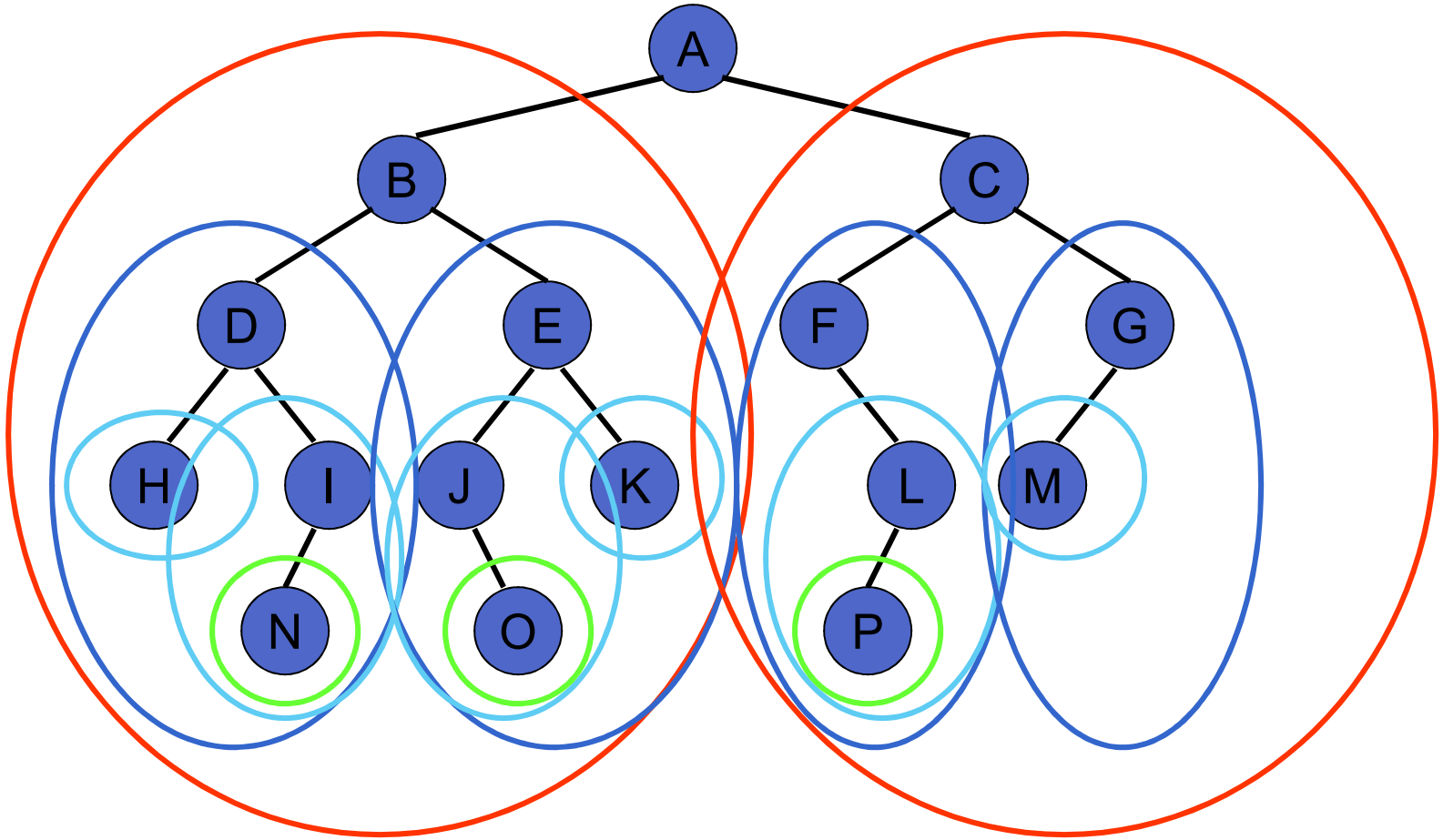
Left - Right – Node

LRN

- Kiểu duyệt này trước tiên thăm các nút của cây con trái sau đó thăm đến cây con phải rồi cuối cùng mới thăm nút gốc (LRN).

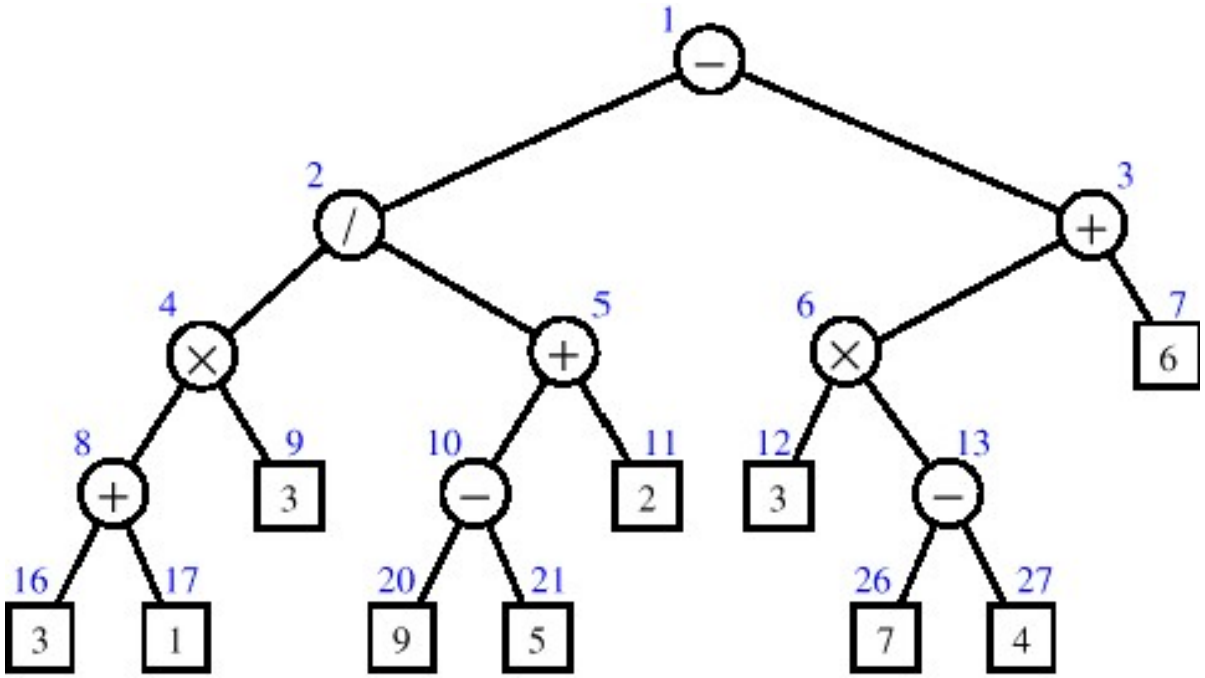
```
void  PosOrder(Tree T)
{
        if (T!=NULL)
        {
                PosOrder(T->left);
                PosOrder(T->right);
                <Xử lî ỉt>
        }
}
```

Duyệt theo thứ tự sau (Left-Right-Node)



Kết quả: H N I D O J K E B P L F M G C A

Duyệt theo thứ tự giữa (Left-Node-Right) (biểu thức dạng trung tố)



(3 + 1)×3/(9 – 5 + 2) – (3×(7 – 4) + 6)

Biểu diễn cây tổng quát bằng cây nhị phân

- Nhược điểm của các cấu trúc cây tổng quát:
  - Bậc của các nút trên cây có thể dao động trong một biên độ lớn $\Rightarrow$ việc biểu diễn gặp nhiều khó khăn và lãng phí.
  - Việc xây dựng các thao tác trên cây tổng quát phức tạp hơn trên cây nhị phân nhiều.
- Vì vậy, thường nếu không quá cần thiết phải sử dụng cây tổng quát, người ta chuyển cây tổng quát thành cây nhị phân.

Biểu diễn cây tổng quát bằng cây nhị phân

- Ta có thể biến đổi một cây bất kỳ thành một cây nhị phân theo qui tắc sau:
  - Giữ lại nút con trái nhất làm nút con trái.
  - Các nút con còn lại chuyển thành nút con phải.
  - Như vậy, trong cây nhị phân mới, con trái thể hiện quan hệ cha con và con phải thể hiện quan hệ anh em trong cây tổng quát ban đầu.

Cây nhị phân tìm kiếm (Binary search tree - BST)

Định nghĩa và ví dụ cây nhị phân tìm kiếm (BST)

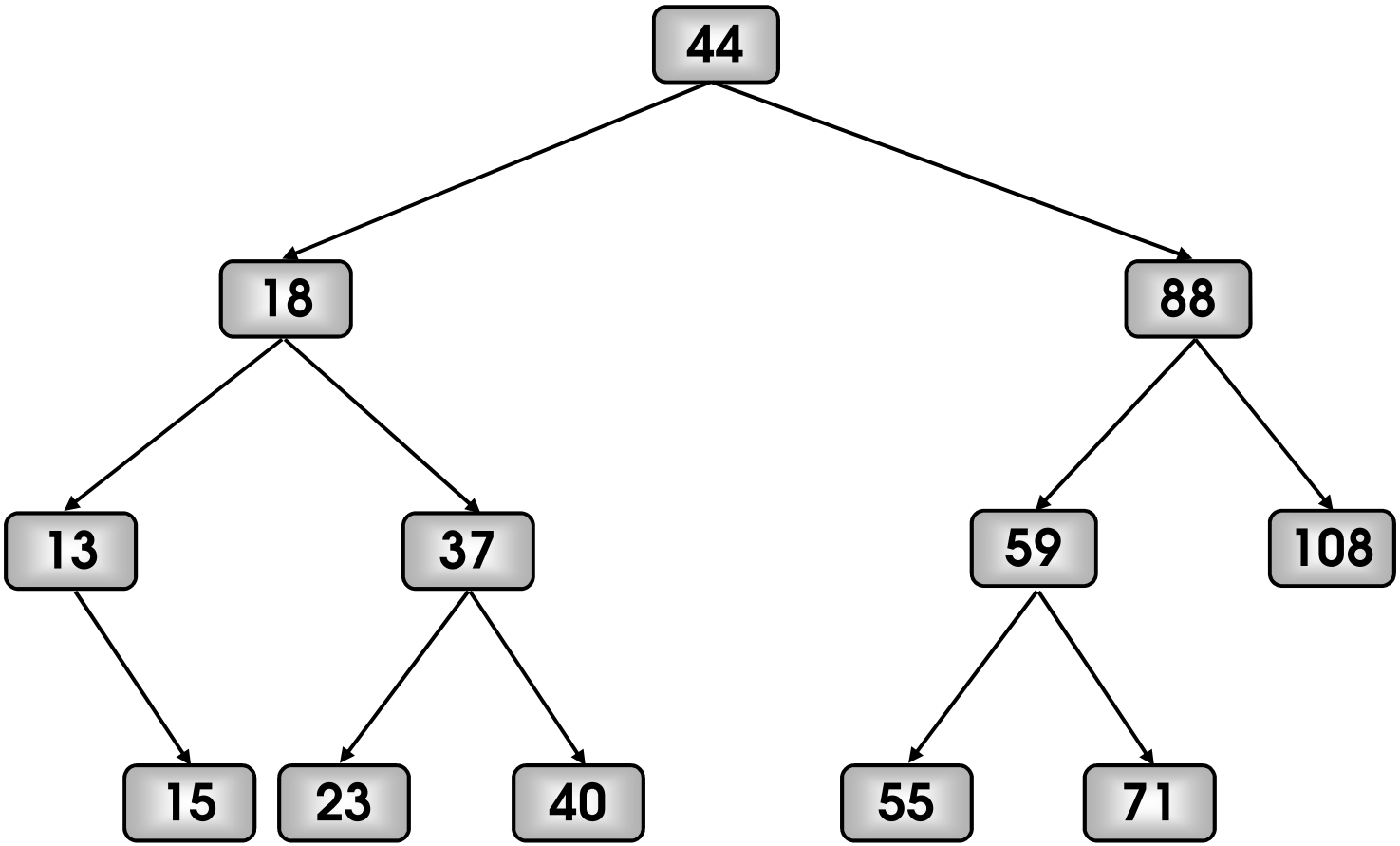
- Định nghĩa:

  Cây nhị phân tìm kiếm (BST) là cây nhị phân trong đó tại mỗi nút, khóa của nút đang xét lớn hơn khóa của tất cả các nút thuộc cây con trái và nhỏ hơn khóa của tất cả các nút thuộc cây con phải.

- Nhận xét :

1. Khóa của các nút phải đôi một khác nhau.
2. Khóa của nút có thể biến đơn hay là một trường dữ liệu nào đó của một biến cấu trúc
3. Khóa của nút phải là các dữ liệu so sánh được như : ký tự, số nguyên, số thực, chuỗi (so sánh theo thứ tự từ điển),...
4. Ngoài ra, đối với các biến cấu trúc như thông tin về nhân viên, sinh viên ,...ta cũng có thể lưu trử trên các nút của BST khi ta dùng các khóa là các trường dữ liệu của cấu trúc so sánh được với nhau như mã nhân viên (sinh viên).
5. Nếu số nút trên cây là N thì chi phí tìm kiếm trung bình chỉ khoảng $\log_2 N$, nên thao tác tìm kiếm trên BST là nhanh

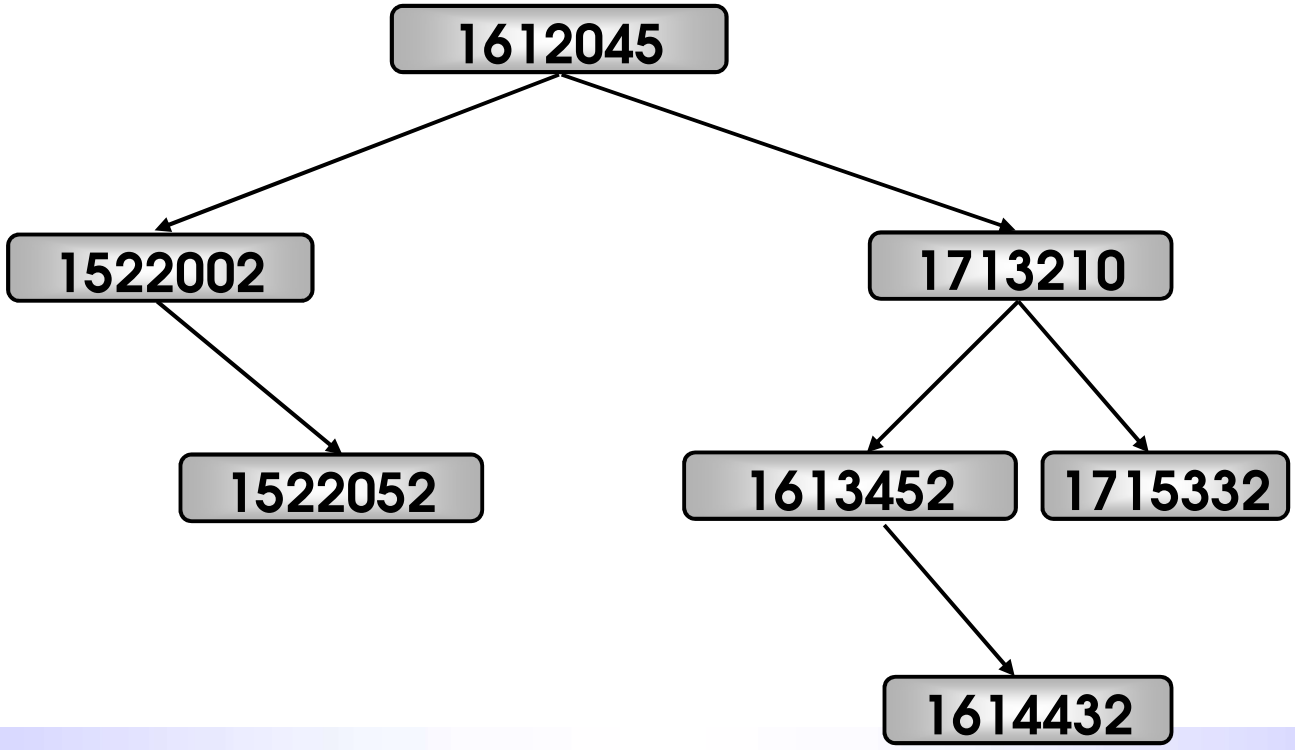
Ví dụ:

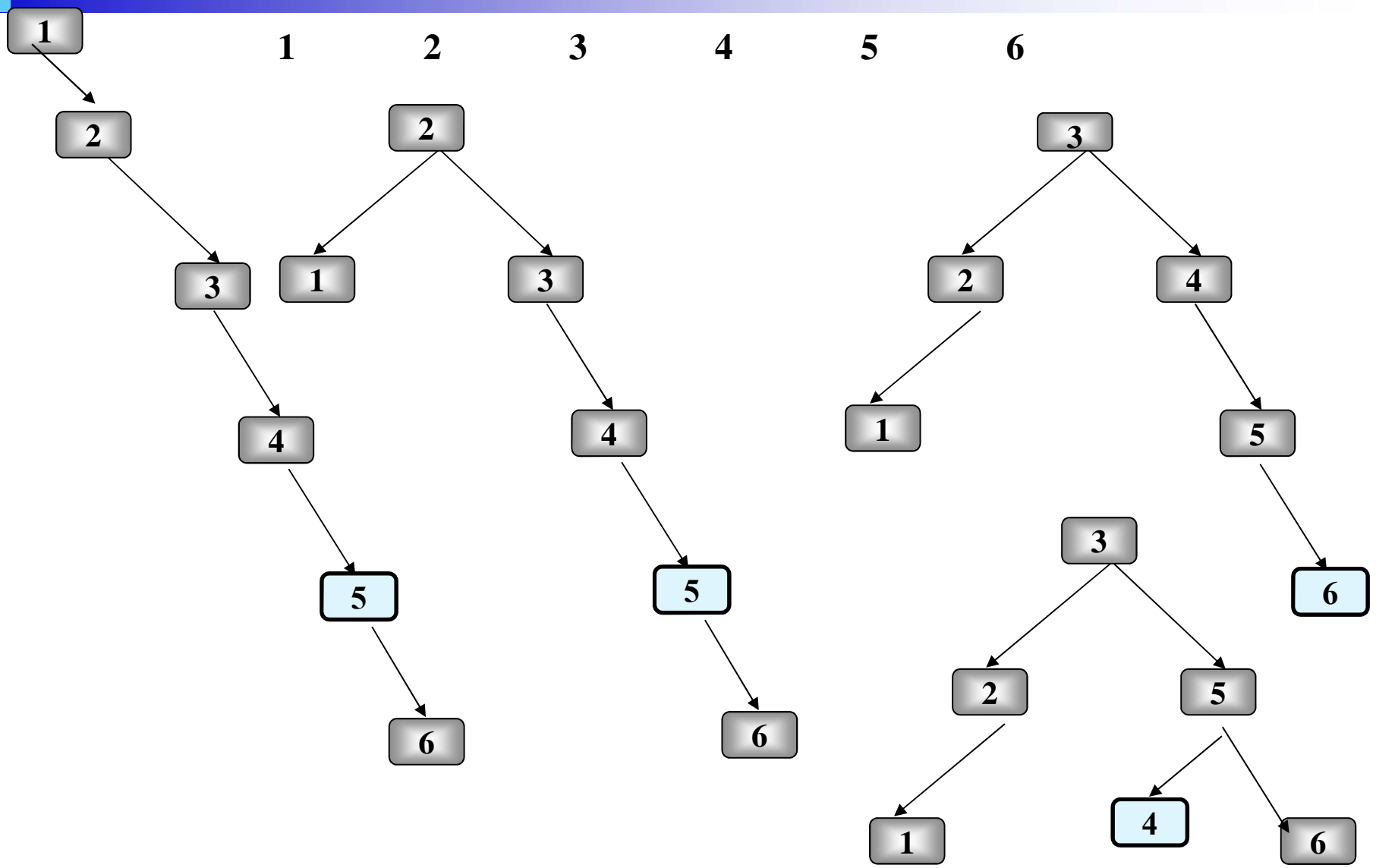


Xem bảng điểm môn học :

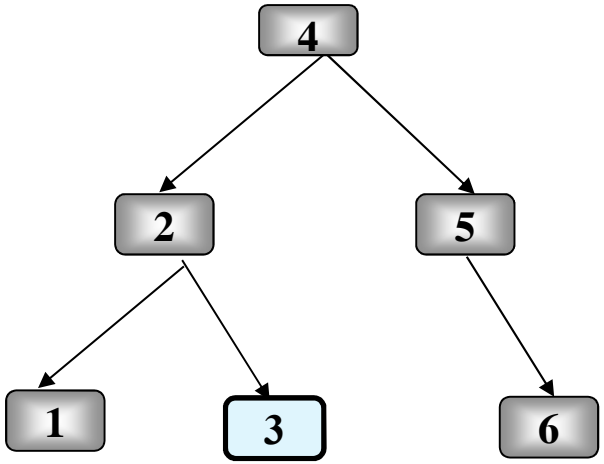
1612045	Nguyen	Tuan	Vo	1	1	1980	CNTT50	9.3
1713210	Ly	Van	Hoa	10	10	1985	CNTT51	6
1613452	Tran	Ngoc	Ninh	5	12	1974	CNTT50	4
1614432	Nguyen		Vo	20	2	1985	CNTT50	6
1715332	Le	Thi	Lieu	2	2	1974	CNTT51	8.7
1522002	Van	Thi	Hoa	25	1	1984	CNTT49	7.2
1522052	Vo	Ngoc	Hoa	10	10	1985	CNTT49	8.7

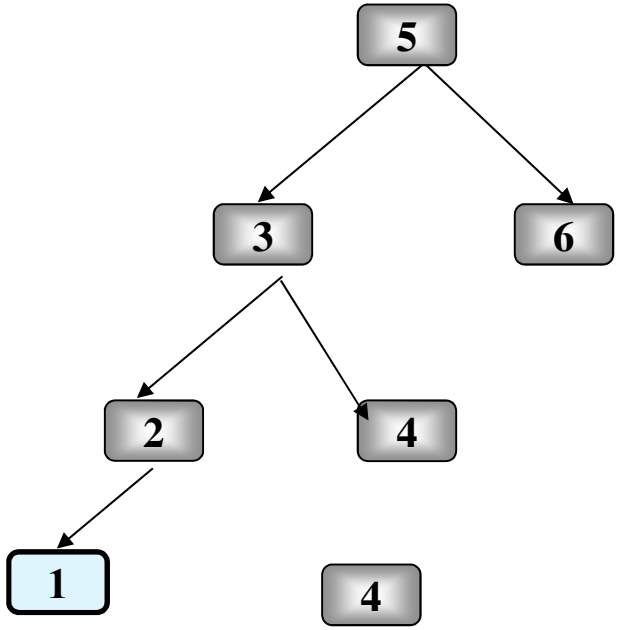
Với dữ liệu trên, ta có thể chuyển vào lưu trử trên cây BST như sau (Khi dùng khóa là trường mã sinh viên) :




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

. . .


4
2
5
1
3
6


5
3
6
2
4
1
4

Cài đặt cấu trúc dữ liệu cây nhị phân tìm kiếm (BST):

- Trường hợp nút có dữ liệu đơn giản : ký tự, số nguyên, số thực
- Trường hợp nút có dữ liệu phức tạp : cấu trúc

Cài đặt cấu trúc dữ liệu cây nhị phân tìm kiếm (BST):

```
//Giả sử kiểu dữ liệu của thành phần dữ liệu của các nút là int.
//Cho nên khóa các nút cũng chính là dữ liệu của  nút, có kiểu int
typedef int  KeyType;
//Kieu cac nut cua cay
struct BSNode
{
        KeyType key;
        BSNode *left;  //Chua dia chi cay con trai
        BSNode *right; //Chua dia chi cay con phai
};

//Kieu CTDL Cay nhi phan tim kiem :
//kieu con tro tro den cac nut kieu BSNode
typedef BSNode *BSTree;
```

Các thao tác trên BST

```
//1. Tạo nút với key x cho trước : CreateNode
//Input x
//Output : NULL ; khong thanh cong
//          p; tro den nut vua tao, neu thanh cong
BSNode  *CreateNode(KeyType x) // ≡ GetNode : tao nut
{
        BSNode *p = new BSNode;
        if (p != NULL)
        {
                p->key = x;
                p->left = NULL;
                p->right = NULL;
        }
        return p;
}
```

2. Tạo cây BST rỗng (khởi tạo)

Cho con trỏ quản lý địa chỉ nút gốc có giá trị NULL

```
//Input : root
//Output : root

void CreateBST(BSTree &root)
{
        root = NULL;
}
```

Thêm một phần tử x vào BST

- Thêm một phần tử x vào cây:
  - Việc thêm một phần tử X vào cây phải bảo đảm điều kiện ràng buộc của BST. ***Ta có thể thêm vào nhiều chỗ khác nhau trên cây***, nhưng nếu thêm vào một nút ngoài sẽ là tiện lợi nhất do ta có thể thực hiện quá trình tương tự thao tác tìm kiếm. Khi chấm dứt quá trình tìm kiếm cũng chính là lúc tìm được chỗ cần thêm.

    (Nút ngoài : là nút ảo không có trên cây, nếu thêm vào vị trí này thì nó sẽ là con của một nút lá).
  - Hàm ***InsertNode*** chèn giá trị x vào cây : trả về giá trị –1 (khi không đủ bộ nhớ); 0 (gặp nút đã có x); 1(thành công) :

Input : x, root;

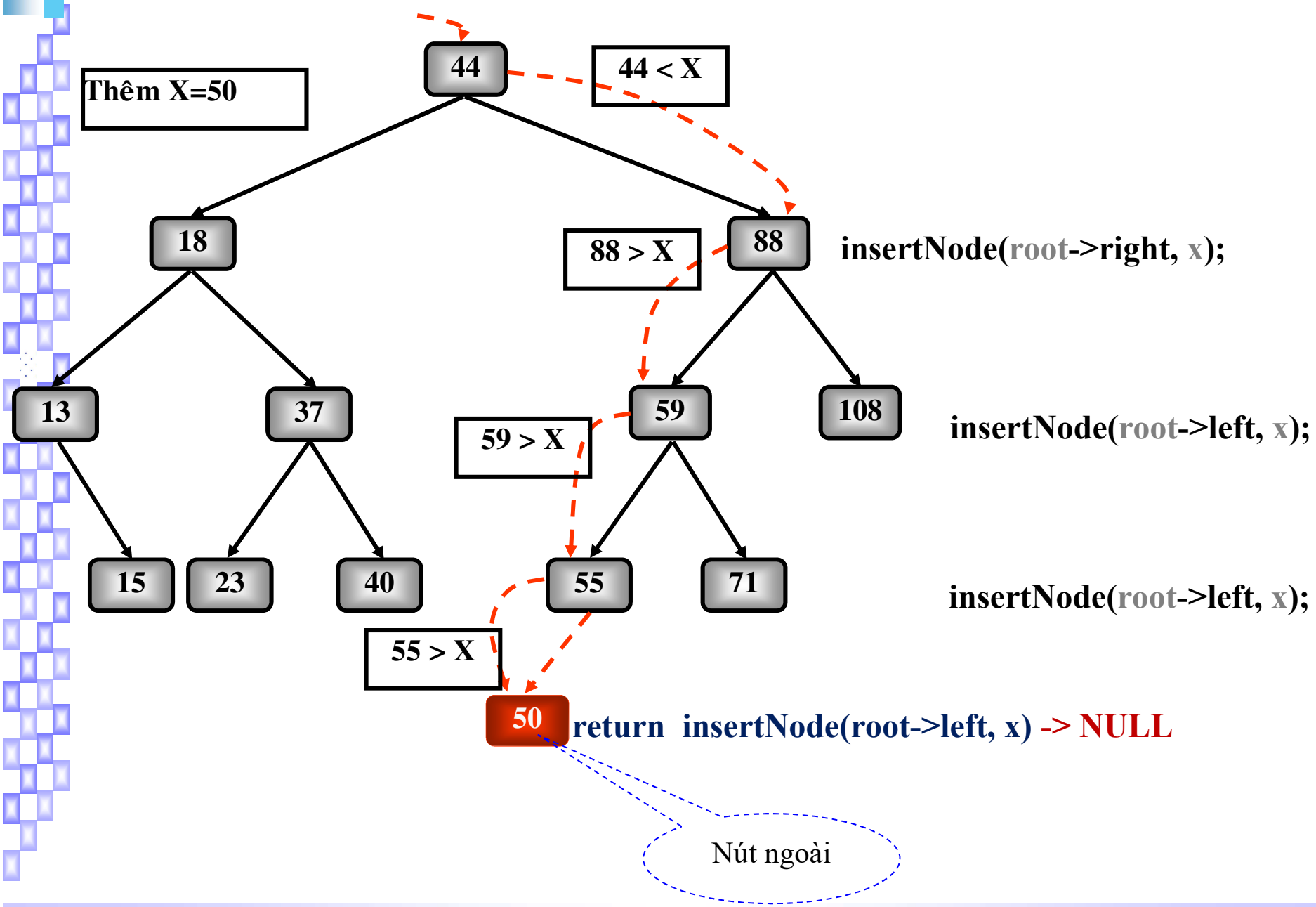
Output : -1; khong thanh cong - khong du bo nho

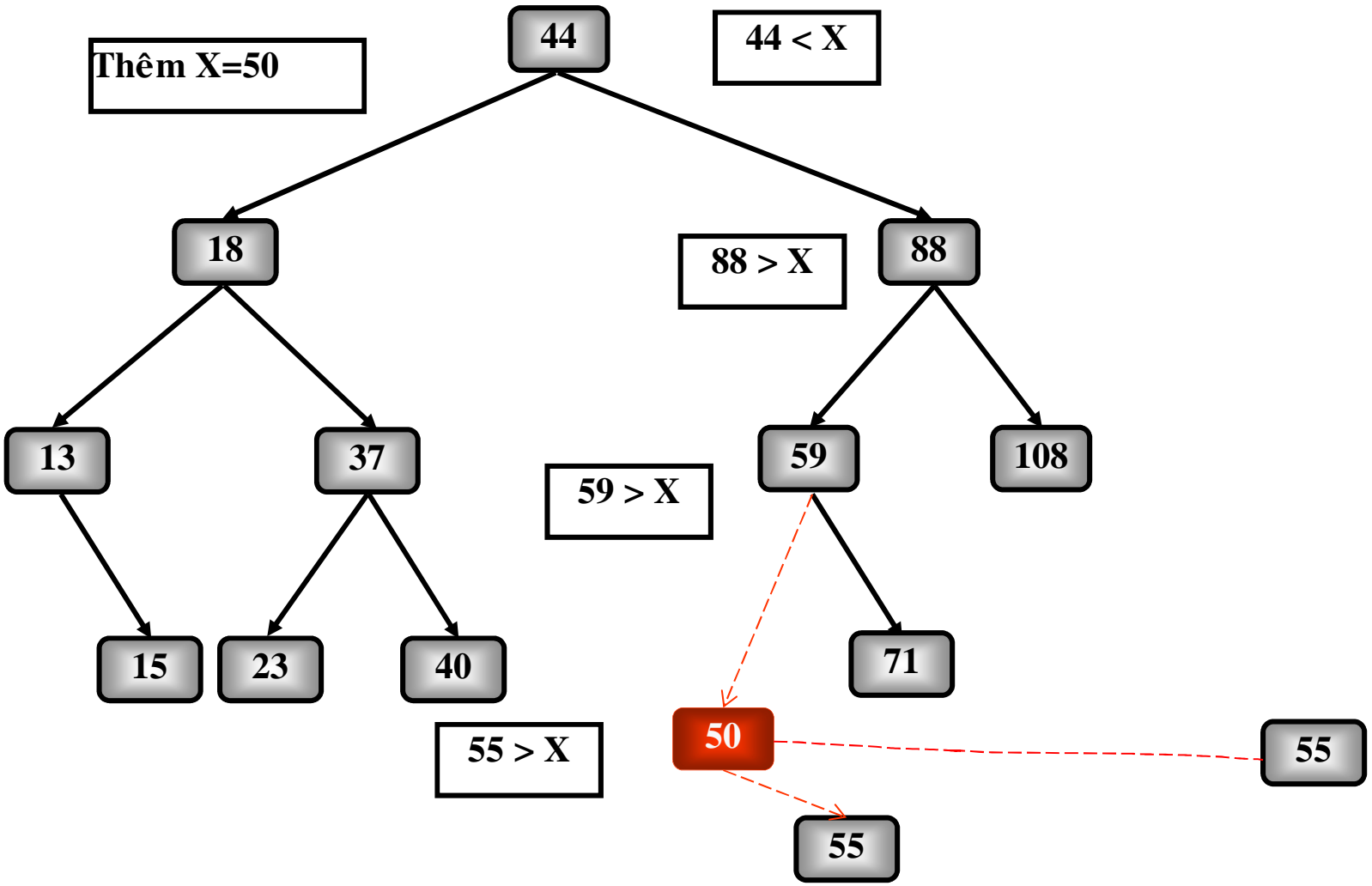
0; khong thanh cong - x da co san trong cay

1; Thanh cong; cay BST moi co them nut chua x

```
int insertNode(BSTree &root, KeyType x)
{
        if (root != NULL) //Cay khac rong
        {
                if (root->key == x)
                        return 0; // x da co san
                if (root->key >  x)
                        return  insertNode(root->left, x);
                else
                        return insertNode(root->right, x);
        }//root == NULL

        root = CreateNode(x);
        if (root == NULL)
                return -1; //khong du bo nho
        return 1; //thanh cong
}
```


Thêm X=50
44
44 < X
18
88 > X
88
insertNode(root->right, x);
13
37
59 > X
59
108
insertNode(root->left, x);
15
23
40
55
71
insertNode(root->left, x);
55 > X
50
return insertNode(root->left, x) -> NULL
Nút ngoài


Thêm X=50
44
44 < X
18
88 > X
88
13
37
59 > X
59
108
15
23
40
71
55 > X
50
55
55

Tạo cây nhị phân tìm kiếm

- Ta có thể tạo một cây nhị phân tìm kiếm bằng cách lặp lại quá trình thêm 1 phần tử vào một cây rỗng.
- Có thể nhập từ bàn phím hay từ tập tin

- Tạo BST bằng cách lặp lại quá trình thêm 1 phần tử nhập từ bàn phím vào một BST rỗng.

```
void CreateBSTree(BStree  &root)
{
        int n, kq, i;
        KeyType x;
        cout<<"Nhap n = "; cin>>n;
        for(i = 0; i < n; i++)
        {
                do
                {
                        cout<<"Nhap gia tri:"; cin>>x;
                        kq = insertNode(root,x)
                        if(kq == -1)
                                return; //khong du bo nho
                } while (kq  == 0); //x co san thi nhap lai
        }
}
```

- Chuyển dữ liệu vào BST từ tập tin :

Giả sử KeyType là kiểu int, tập tin gồm các số nguyên kiểu int. Ta viết hàm chuyển dữ liệu từ tập tin vào cây BST như sau:

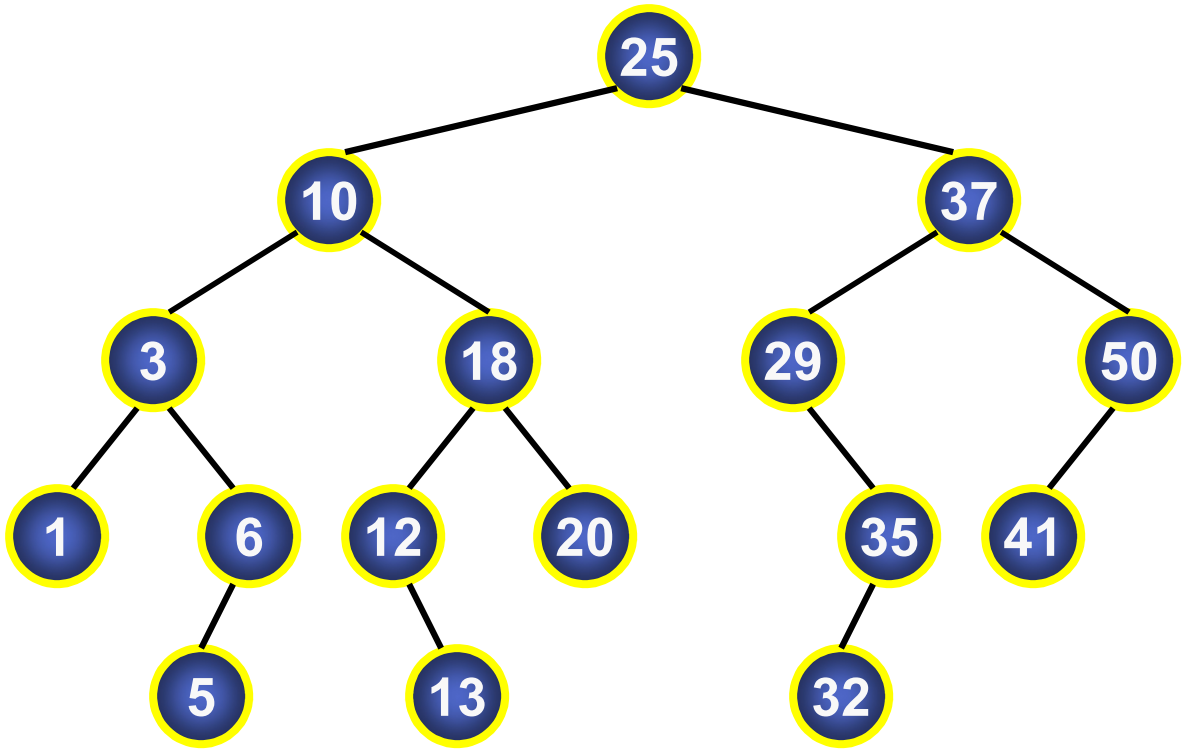
Input : filename

Output : 0; khong thanh cong

1; thanh cong

```
int File_BST(BSTree &root, char *filename)
{
        ifstream in(filename);
        if (!in)
                return 0;
        KeyType x;
        int kq;
        CreateBST(root);
        in >> x;
        kq = insertNode(root, x);
        if (kq == 0 || kq == -1)
                return 0;
        while (!in.eof())
        {
                in >> x;
                kq = InsertNode(root, x);
                if (kq == 0 || kq == -1)
                        return 0;
        }
        in.close();
        return 1;
}
```

25 37 10 18 29 50 3 1 6 5 12 20 35 13 32 41



25	37	10	18	29	50	3	1	6	5	12	20	35	13	32	41

Duyệt cây nhị phân tìm kiếm

- Thao tác duyệt cây trên cây nhị phân tìm kiếm hoàn toàn giống như trên cây nhị phân.
- Lưu ý: khi duyệt theo thứ tự giữa, trình tự các nút duyệt qua sẽ cho ta một dãy các nút theo thứ tự tăng dần của khóa.

- Duyệt theo thứ tự trước – PreOrder - **(Node-*Left*-*Right*):**

Duyệt nút gốc, duyệt cây con bên trái, duyệt cây con bên phải

```
void PreOrder(BSTree root)
{
        if (root != NULL)
        {
                //<Xử lý nút : xuất ra màn hình>
                cout << root->key << '\t';
                PreOrder(root->left);
                PreOrder(root->right);
        }
}
```

- Duyệt theo thứ tự giữa – InOrder - **(*Left*-Node-*Right*):**

Duyệt cây con bên trái, duyệt nút gốc, duyệt cây con bên phải

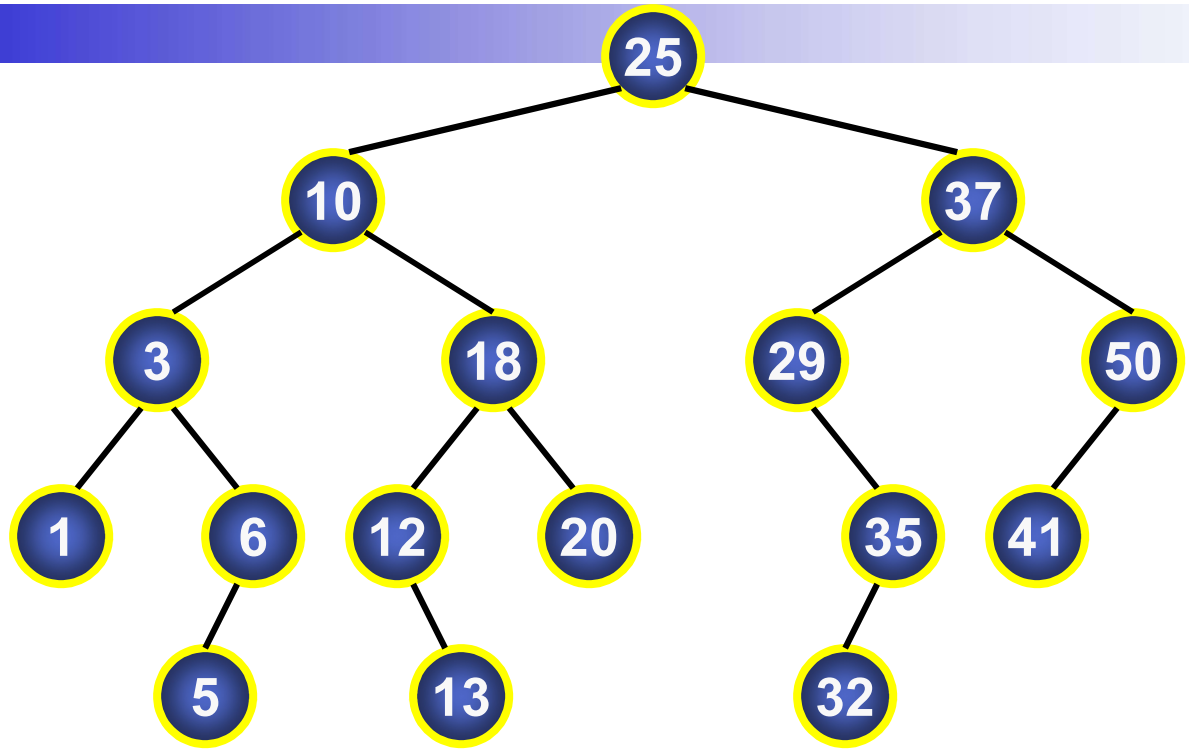
```
void InOrder(BSTree root)
{
        if (root != NULL)
        {
                InOrder(root->left);
                 //<Xử lý nút : xuất ra màn hình>
                cout << root->key << '\t';
                InOrder(root->right);
        }
}
```

Lưu ý : Xuất theo thứ tự giữa ta sẽ có danh sách dữ liệu các nút tăng

- Duyệt theo thứ tự sau – PosOrder - **(*Left*-*Right*-Node):**

Duyệt cây con bên trái, duyệt nút gốc, duyệt cây con bên phải

```
void PosOrder(BSTree root)
{
        if (root != NULL)
        {
                PosOrder(root->left);
                PosOrder(root->right);
                 //<Xử lý nút : xuất ra màn hình>
                cout << root->key << '\t';
        }
}
```



25	37	10	18	29	50	3	1	6	5	12	20	35	13	32	41

```
Cay BST hien hanh, xuat theo thu tu truoc (NLR) :
25        10        3         1         6         5         18        12        13        20
37        29        35        32        50        41

Cay BST hien hanh, xuat theo thu tu giua (LNR) :
1         3         5         6         10        12        13        18        20        25
29        32        35        37        41        50

Cay BST hien hanh, xuat theo thu tu cuoi (LRN) :
1         5         6         3         13        12        20        18        10        32
35        29        41        50        37        25
```

Thuật toán sắp trên cây sắp tăng dần một dãy.

Thực hiện các bước :

- Đưa dãy lên cây BST.
- Duyệt theo thứ tự giữa (Inorder : LNR) để xuất cây ra màn hình,
- Kết quả : ta có dãy tăng.

Tìm một phần tử x trong cây (đệ quy)

Input : x, root
Output : p, con tro tro den nut chua x neu co
NULL, neu khong co

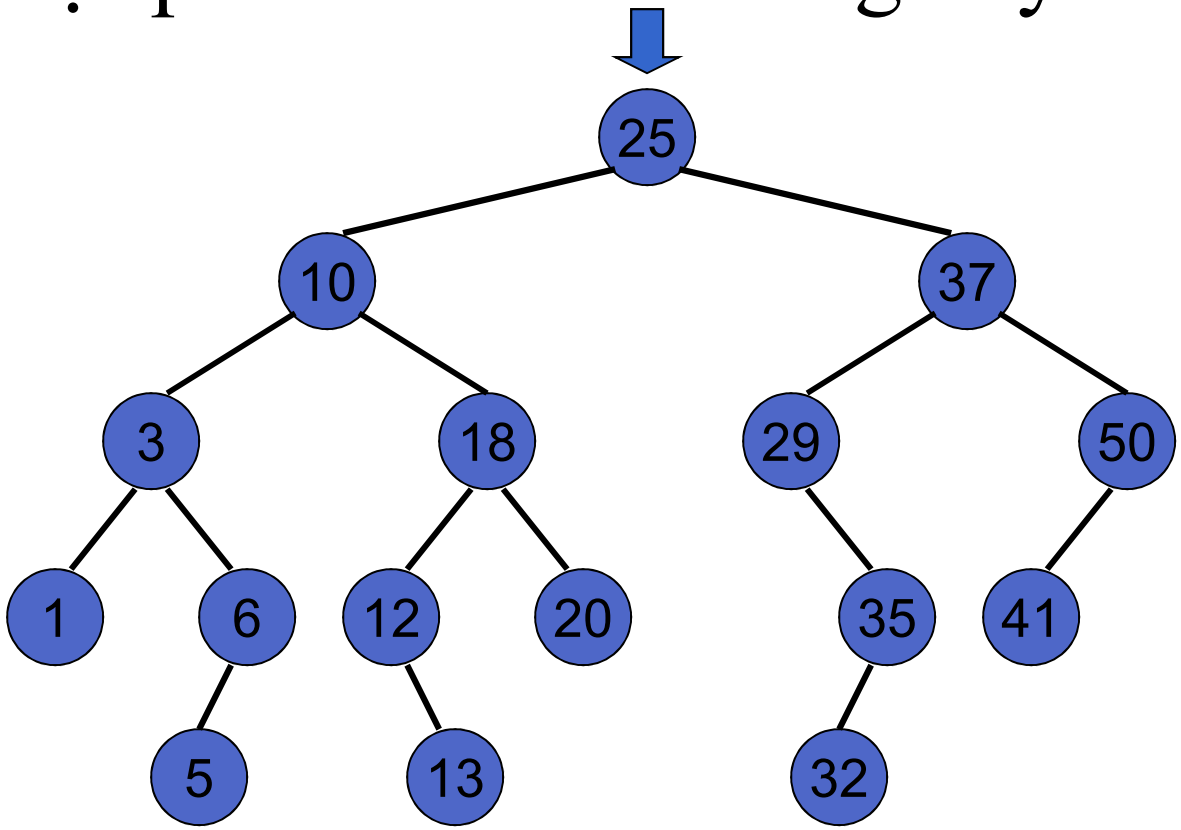
```
BSTree Search(KeyType x, BSTree root)
{
    if (root != NULL)
    {
        if (root->key == x) //Tim thay x
          return root;
        else
            if (root->key < x)
                return Search(x, root->right); //tim x trong cay con phai
            else
                return Search(x, root->left); //tim x trong cay con trai
    }
    return NULL;
}
```

Tìm một phần tử x trong cây (không đệ quy)

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

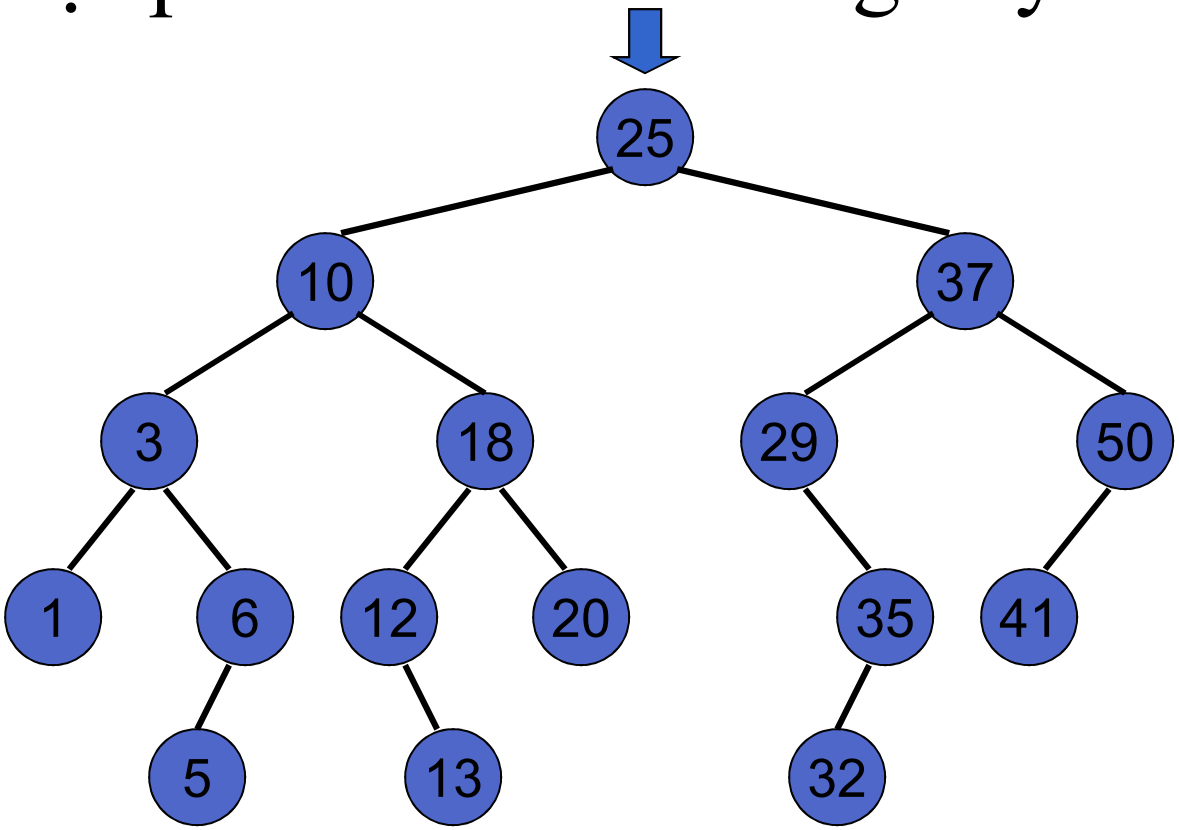
```
BSTree   SearchLap(BSTree   root, KeyType  x, BSTree   &parent)
{
    BSTree  locPtr = root;
    parent = NULL;
    while (locPtr != NULL)
    {
        if (x == locPtr -> key)
                return  locPtr;
        else
        {
                parent = locPtr;
                if (x > locPtr->key)
                        locPtr = locPtr->right;
                 else
                        locPtr = locPtr->left;
        }
    }
    return NULL;
}
```

Tìm một phần tử x=13 trong cây



Tìm kiếm 13 Tìm thấy

Tìm một phần tử x=13 trong cây



Tìm kiếm 13

Tìm thấy

Số node duyệt: 5
Số lần so sánh: 5

Tìm một phần tử x trong cây

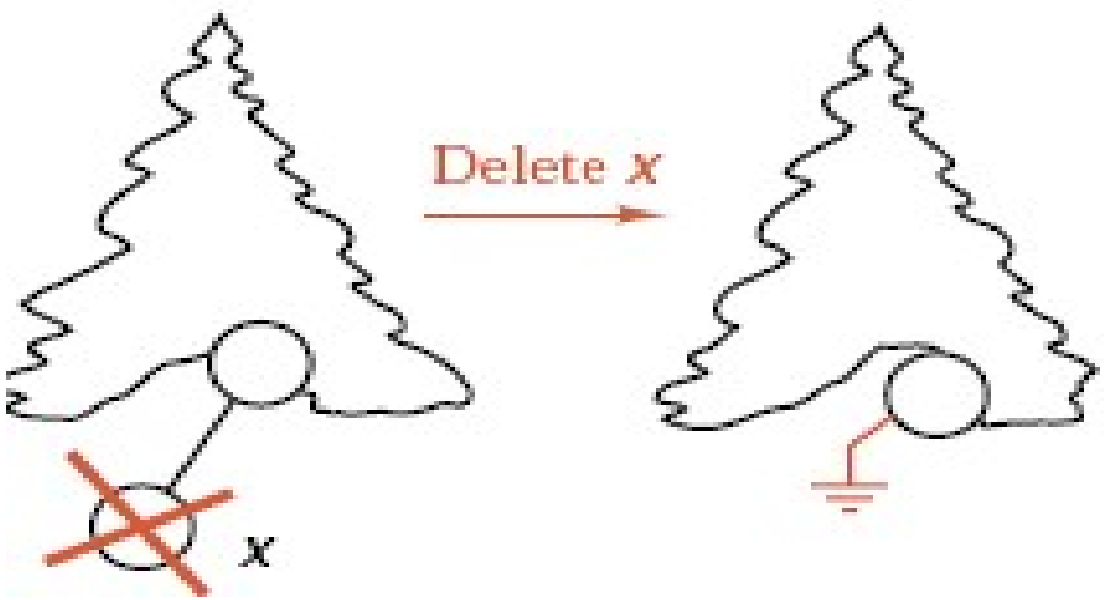
Nhận xét:

- Số lần so sánh tối đa phải thực hiện để tìm phần tử X là h, với h là chiều cao của cây.
- Như vậy thao tác tìm kiếm trên BST có n nút tốn chi phí trung bình khoảng $O(\log_2 n)$.

Hủy một phần tử có khóa x

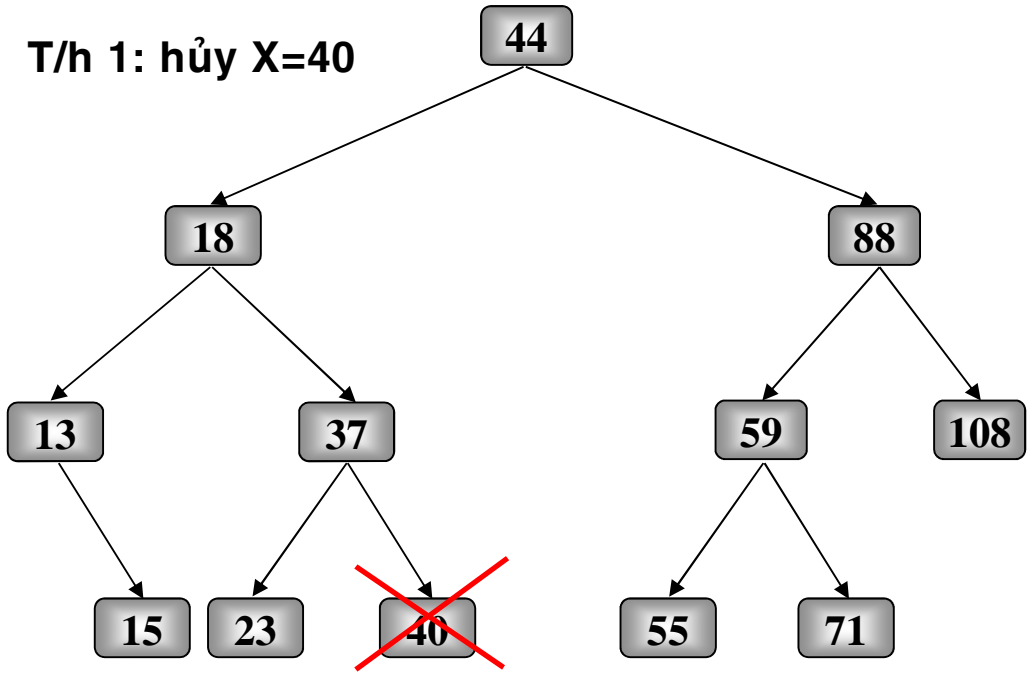
- Việc hủy một phần tử X ra khỏi cây phải bảo đảm điều kiện ràng buộc của BST.
- Có 3 trường hợp khi hủy nút X có thể xảy ra:
  - X là nút lá (không con)
  - X chỉ có 1 con (trái hoặc phải).
  - X có đủ cả 2 con

Trường hợp 1: X là nút lá.

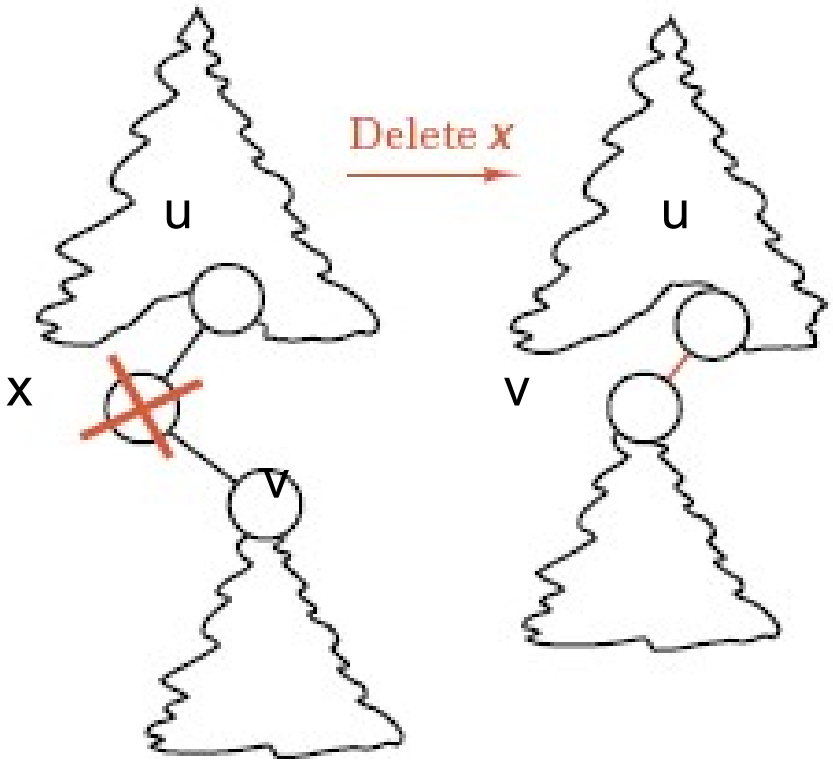


1. Xóa node này
2. Gán liên kết từ cha của nó thành rỗng

- **Ví dụ :** chỉ đơn giản hủy X vì nó không móc nối đến phần tử nào khác.

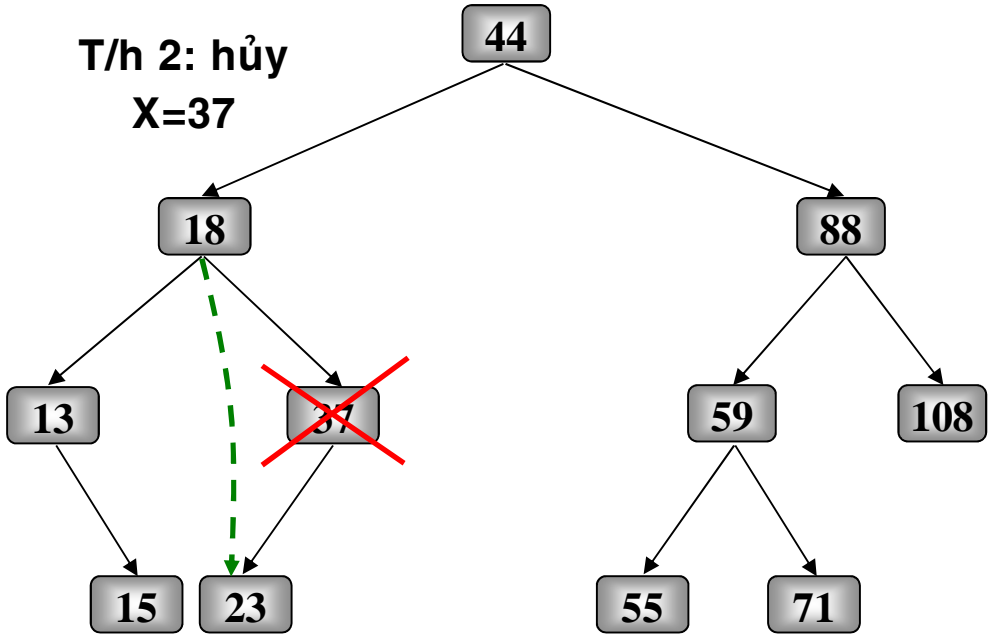


Trường hợp 2: X chỉ có 1 con (trái hoặc phải)



1. **Gán liên kết từ cha của nó xuống con duy nhất của nó**
2. **Xóa node này.**

- **Ví dụ :** Trước khi hủy X = 37, ta móc nối cha của X là 18 với con duy nhất của nó là 23



Trường hợp 3: X có đủ 2 con

- Trong trường hợp này:
  - Không hủy trực tiếp được.
  - Thực hiện hủy gián tiếp theo cách :
    - Thay vì hủy X, ta sẽ tìm một phần tử Y gọi là phần tử thế mạng của X. Phần tử Y này có tối đa một con.
    - Thông tin lưu tại Y sẽ được chuyển lên lưu tại X.
    - Sau đó, nút bị hủy thật sự sẽ là Y giống như 2 trường hợp đầu.

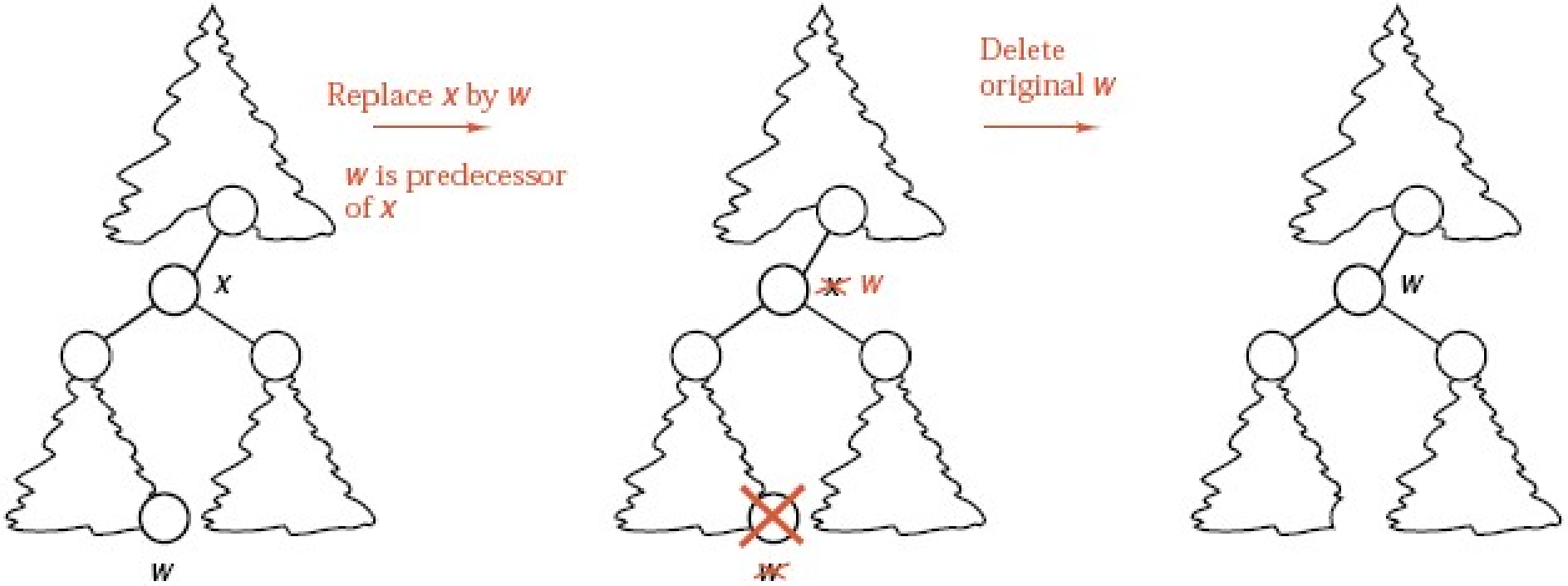
- ***Vấn đề là phải chọn Y sao cho khi lưu Y vào vị trí của X, cây vẫn là BST ?***
- Có 2 phần tử thỏa mãn yêu cầu:
  - ***Phần tử lớn nhất (phải nhất) trên cây con trái.***
  - ***Phần tử nhỏ nhất (trái nhất) trên cây con phải.***
- Việc chọn lựa phần tử nào là phần tử thế mạng hoàn toàn phụ thuộc vào ý thích của người lập trình.

- Hủy nút bằng cách sử dụng nút thế mạng là :

Phần tử lớn nhất trên cây con trái

(tức là phần tử cực phải của cây con trái của x)

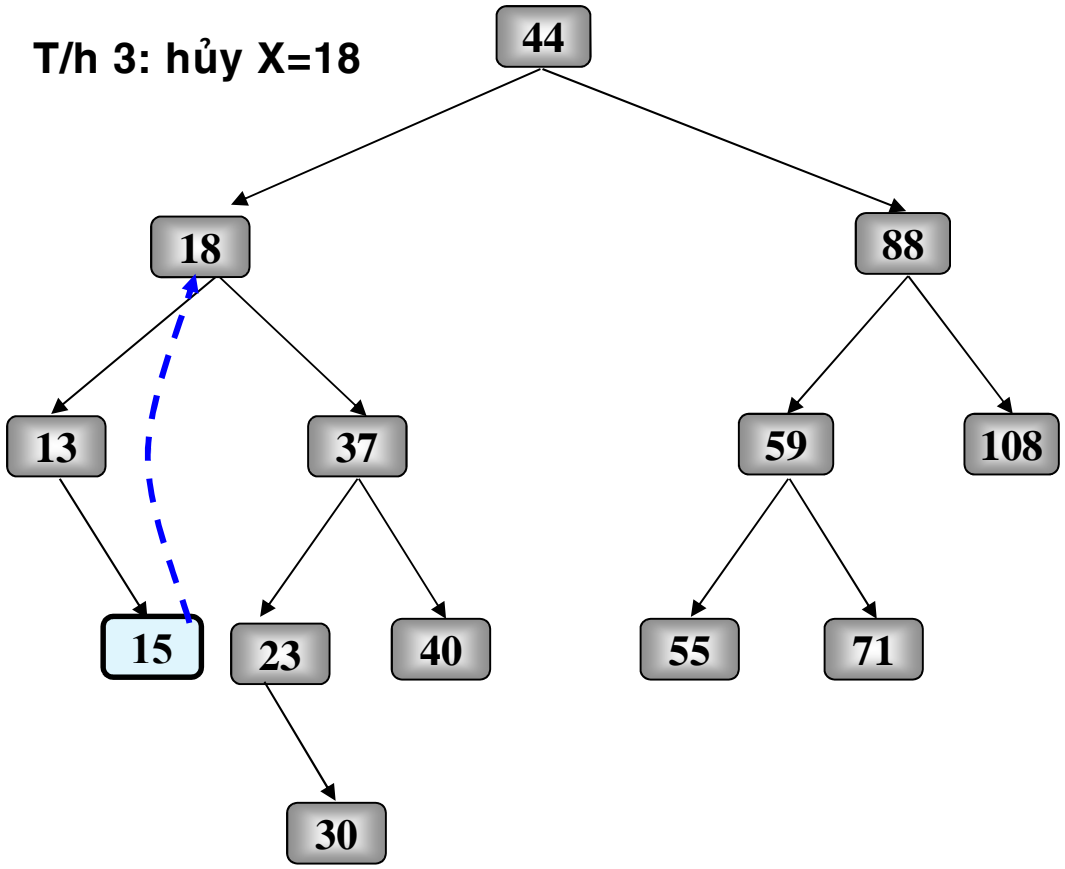
- Chọn ***phần tử lớn nhất (cực phải) trên cây con trái*** làm phần tử thế mạng.



1. Tìm w là ***nút trước nút x trên phép duyệt cây inorder (LNR)*** - chính là nút cực phải của cây con bên trái của x.
2. Thay x bằng w
3. Xóa nút w cũ (giống trường hợp 1 hoặc 2 đã xét)

- Hủy nút 18 :

Chọn phần tử thế mạng là phần tử lớn nhất (cực phải) của cây con trái : là nút 15 (là ***nút kế trước nút x(18) trên phép duyệt cây inorder (LNR))***



13 15 18 23 30 37 40 44 55 59 71 88 108

```
//Tim nut co gia tri lon nhat cua cay con trai cua root
//Input root
//Output : Gia tri nut co gia tri lon nhat cua cay con trai
KeyType DeleteMax(BSTree &root)
{
    KeyType k;
    if (root->right == NULL)
    {
        k = root->key;
        root = root->left;
        return k;
    }
    else
        return DeleteMax(root->right);
}
```

//Huy mot nut co khoa cho truoc ra khoi cay : nut the mang la phan tu lon nhat cua cay cay con trai
//Input : x, root
//Output : 1. root (cay BST ket qua root) neu thanh cong
//0; khong thanh cong

```
int DeleteNode(KeyType  x, BSTree &root)
{
    if (root != NULL)
    {
        if (x < root->key)
            return DeleteNode(x, root->left);
        else
        if (x  > root->key)
            return DeleteNode(x, root->right);
        else //x  == root->key
        if ((root->left == NULL) && (root->right == NULL)) //khong co con trai, khong co con phai
           root = NULL;
        else
            if (root->left == NULL) //co 1 con : con phai, khong co con trai
                root = root->right;
            else
                if (root->right == NULL) //co 1 con : con trai, khong co con phai
                   root = root->left;
                else //co ca 2 con trai, phai
                   root->key = DeleteMax(root->left); //Huy nut co gia tri lon nhat cua cay con trai x
          return 1; //Thanh cong
    }
    return 0;
}
```

- Hủy nút bằng cách sử dụng :

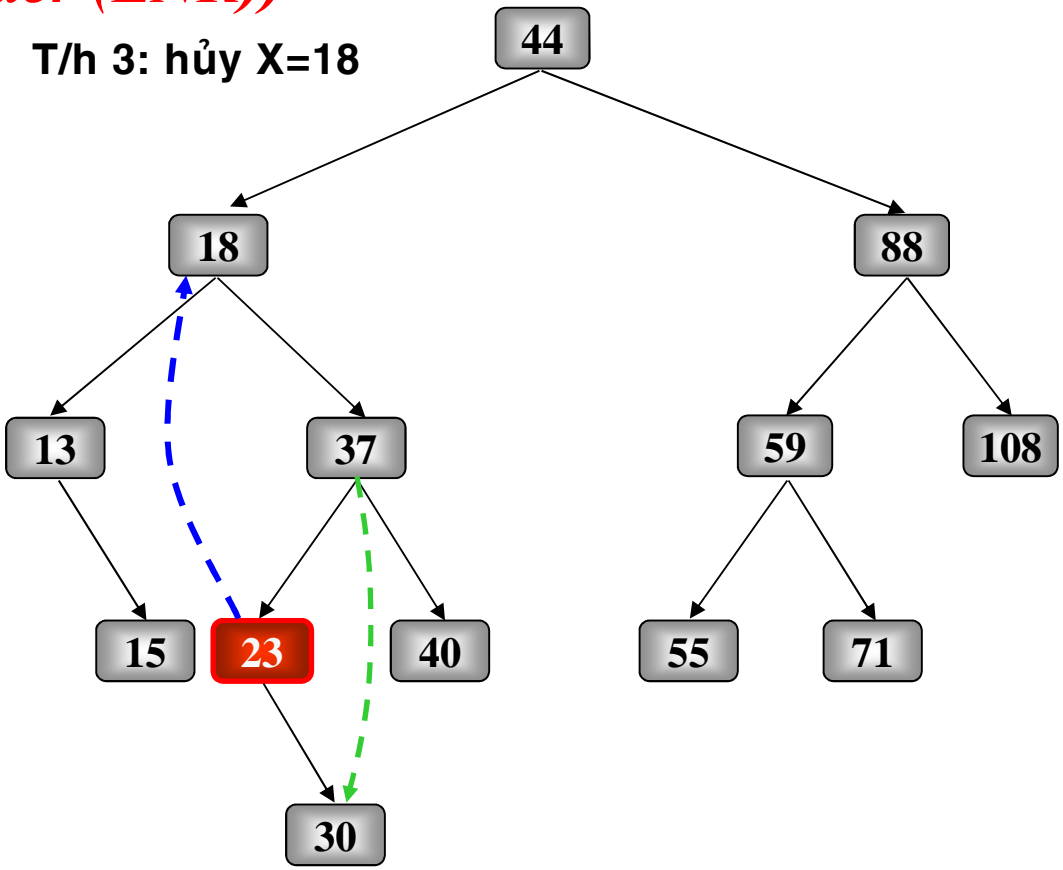
Nút thế mạng là ***Phần tử nhỏ nhất trên cây con phải (tức là phần tử cực trái của cây con phải của x)***

- Chọn ***phần tử nhỏ nhất (cực trái) trên cây con phải*** làm phần tử thế mạng.

1. Tìm w là ***nút ké nút x trên phép duyệt cây inorder (LNR)*** - chính là nút cực trái của cây con bên phải của x.
2. Thay x bằng w
3. Xóa nút w cũ (giống trường hợp 1 hoặc 2 đã xét)

Ví dụ : Hủy X = 18

- Chọn phần tử thế mạng là phần tử nhỏ nhất (trái nhất) của cây con phải : Phần tử 23. (là ***nút kế sau nút x trên phép duyệt cây inorder (LNR))***



13 15 18 23 30 37 40 44 55 59 71 88 108

```
//Tim Nut the mang : Nut co gia tri nho nhat cua cay con phai cua root
//Input root
//Output : Gia tri Nut co gia tri nho nhat cua cay con phai

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

❖ **Huy mot nut co khoa cho truoc ra khoi cay**

Input root, x

Output : 1, root; cây BST kết quả, đã xóa được nút có x neu thanh cong

0; khong thanh cong

```
int DeleteNode(KeyType  x, BSTree &root)
{
        if (root != NULL)
        {
                if (x < root->key)
                  return DeleteNode(x, root->left);
                else
                     if (x  > root->key)
                          return DeleteNode(x, root->right);
                     else //x  == root->key
                          if ((root->left == NULL) && (root->right == NULL)) //khong co con trai, khong co con phai
                                  root = NULL;
                              else
                                  if (root->left == NULL) //co 1 con : con phai, khong co con trai
                                      root = root->right;
                                  else
                                      if (root->right == NULL) //co 1 con : con trai, khong co con phai
                                              root = root->left;
                                      else //co ca 2 con trai, phai
                                              root->key = DeleteMin(root->right); //Nut trái nhất cây con phải của x
                return 1; //thanh cong
        }
        return 0; //khong thanh cong
}
```

Hủy toàn bộ cây nhị phân tìm kiếm

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

```
void RemoveTree(BSTree  &root)
{
   if(root)
   {
      RemoveTree(root->left);
      RemoveTree(root->right);
      delete(root); //root = NULL;
   }
}
```

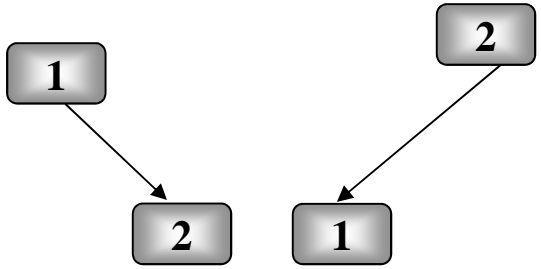
Cây nhị phân tìm kiếm

Nhận xét:

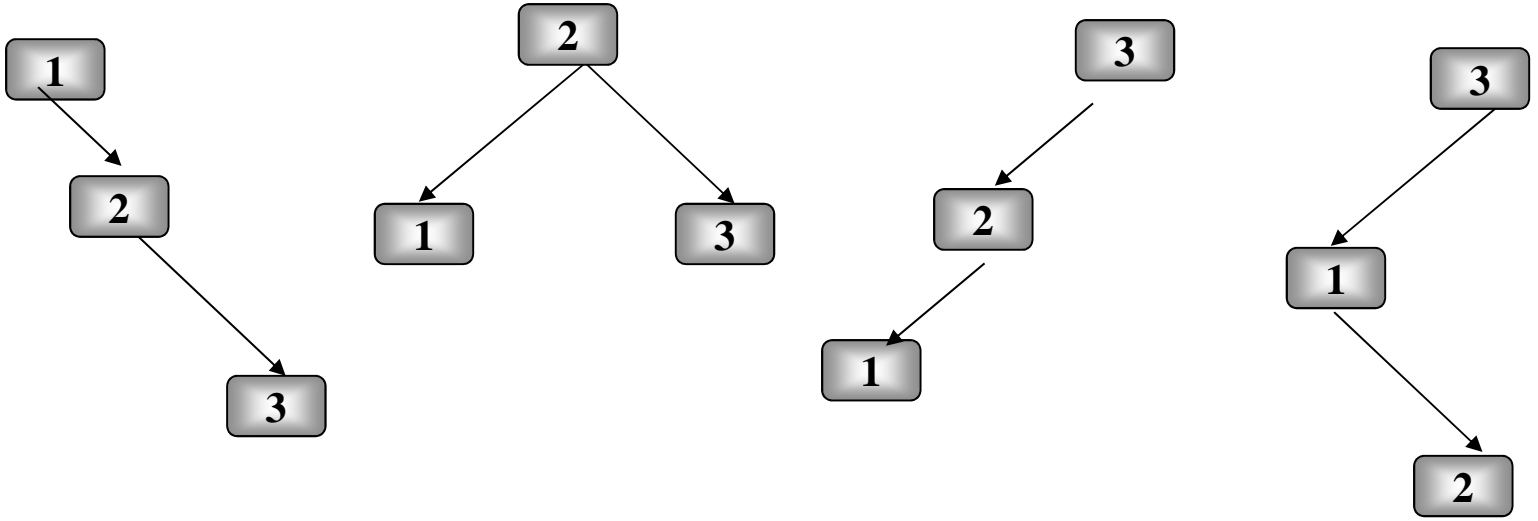
- Tất cả các thao tác searchNode, InsertNode, DeleteNode đều có độ phức tạp trung bình O(h), với h là chiều cao của cây
- Trong trong trường hợp tốt nhất, BST có n nút sẽ có độ cao h = $\log_2(n)$. Chi phí tìm kiếm khi đó sẽ tương đương tìm kiếm nhị phân trên mảng có thứ tự.
- Trong trường hợp xấu nhất, cây có thể bị suy biến thành 1 danh sách liên kết (khi mà mỗi nút đều chỉ có 1 con trừ nút lá). Lúc đó các thao tác trên sẽ có độ phức tạp O(n).
- Vì vậy cần có cải tiến cấu trúc của BST để đạt được chi phí cho các thao tác là $\log_2(n)$.

Cây nhị phân cân bằng
(AVL tree)
Adelson-Velskii & Landis

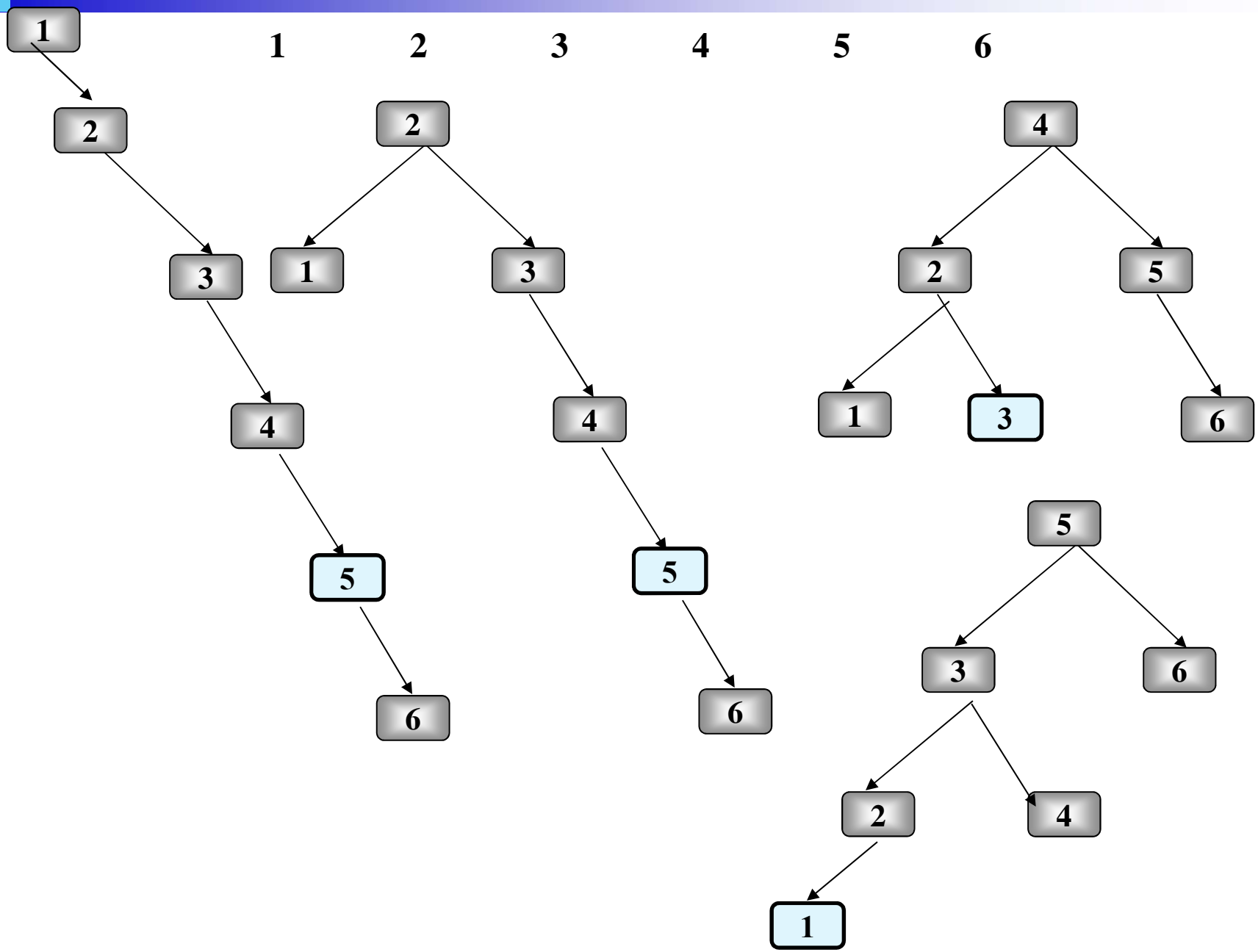
- n = 1 - > 1 BST
- n = 2 - > 2 BST



- n = 3 - > 4 BST



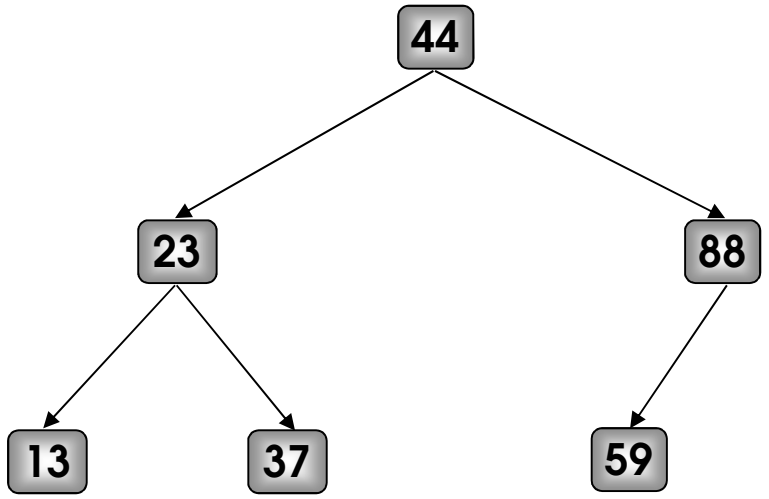
. . .Tổng quát, n nút sẽ liệt kê được 2^{n-1} BST


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

Cây nhị phân cân bằng hoàn toàn

- Định nghĩa:

  Cây cân bằng hoàn toàn là cây nhị phân tìm kiếm mà ***tại mỗi nút của nó***, số nút của cây con trái ***chênh lệch không quá một*** so với số nút của cây con phải.



- Cây cân bằng hoàn toàn n nút sẽ có chiều cao h ≈ $\log_2 n$, nên việc tìm kiếm trên cây cân bằng hoàn toàn sẽ rất nhanh; trong trường hợp xấu nhất ta chỉ phải tìm qua $\log_2 n$ phần tử (n là số nút trên cây).
- Một cây rất khó đạt được trạng thái cân bằng hoàn toàn. Mà một cây cân bằng hoàn toàn cũng rất dễ mất cân bằng khi thêm hay hủy các nút trên cây.
- Một khi cây mất cân bằng, ta có thể cân bằng lại cây !!

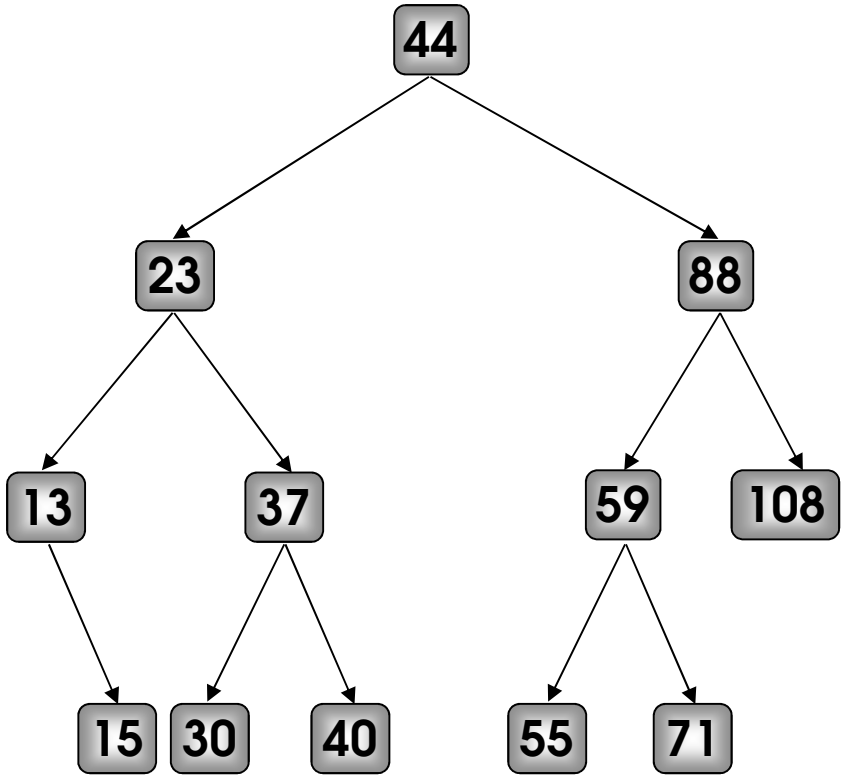
  Khi cân bằng lại cây, cần chi phí lớn vì phải thao tác trên toàn bộ cây.

- Do CCBHT là một cấu trúc kém ổn định nên trong thực tế không thể sử dụng. Nhưng ưu điểm của nó lại rất quan trọng. Vì vậy, cần tìm ra một CTDL khác có đặc tính giống CCBHT nhưng ổn định hơn.
- Như vậy, cần tìm cách tổ chức một cây đạt trạng thái cân bằng yếu hơn nhưng vẫn phải bảo đảm chi phí cho thao tác tìm kiếm đạt ở mức $O(\log_2 n)$.

Cây nhị phân cân bằng (AVL)

Định nghĩa:

Cây nhị phân tìm kiếm cân bằng (AVL) là cây mà tại mỗi nút của nó **chiều cao** *của cây con trái và của cây con phải chênh lệch không quá một.*



Không phải lệch về số nút mà lệch về chiều cao của cây

Cây nhị phân cân bằng hoàn toàn

Cây nhị phân cân bằng

- CCBHT là cây cân bằng. Điều ngược lại không đúng.
- Cây cân bằng là CTDL ổn định hơn CCBHT.

Cây AVL

- Lịch sử cây cân bằng (AVL Tree):
  - AVL là tên viết tắt của các tác giả người Nga đã đưa ra định nghĩa của cây cân bằng Adelson-Velskii và Landis (1962).
  - Từ cây AVL, người ta đã phát triển thêm nhiều loại CTDL hữu dụng khác như cây đỏ-đen (Red-Black Tree), B-Tree, ...
- Cây AVL có chiều cao $h = \log_2(n)$
- Tính cân bằng của cây -> (xét) cân bằng tại các nút.

  Tính cân bằng tại mỗi nút dựa vào khái niệm :

  Chỉ số cân bằng tại mỗi nút.

Cây AVL

Chỉ số cân bằng của một nút trong cây nhị phân tìm kiếm:

- Định nghĩa: Chỉ số cân bằng (CSCB) của một nút trong cây nhị phân tìm kiếm là hiệu số của chiều cao cây con phải và cây con trái của nó.

  CSCB(p) = Chiều cao cây con phải (p) - Chiều cao cây con trái (p)

- Đối với một cây cân bằng, chỉ số cân bằng (CSCB) của mỗi nút chỉ có thể mang một trong ba giá trị sau đây:
  - CSCB(p) = 0 $\Leftrightarrow$ Chiều cao cây con trái (p) = Chiều cao cây con phải (p)
  - CSCB(p) = 1 $\Leftrightarrow$ Chiều cao cây con trái (p) < Chiều cao cây con phải (p)
  - CSCB(p) =-1$\Leftrightarrow$ Chiều cao cây con trái (p) > Chiều cao cây con phải (p)

Cây AVL

- Ký hiệu :
  - Chỉ số cân bằng tại nút p : CSCB(p) = balFactor(p);
  - Chiều cao cây trái (p) : $h_L(p)$
  - Chiều cao cây phải(p) : $h_R(p)$

balFactor(p) = h_R (p)– h_L (p)

- Trên cây AVL, tại mỗi nút p:

balFactor(p) = 1; Cây con phải cao hơn

= -1; Cây con trái cao hơn

= 0; Cây con trái, phải cao bằng nhau

Cài đặt Cây AVL

```
//Định nghĩa các giá trị hằng
#define     LH      -1      /* Cây con trái cao hơn    */
#define     EH       0      /* Hai cây con bằng nhau */
#define     RH       1      /* Cây con phải cao hơn   */
//Định nghĩa kiểu dữ liệu khóa của các nút
typedef <Kieu khoa cua cac nut>  DataType;
//Định nghĩa kiểu các nút
struct tagAVLNode //Kieu cac nut
{
    int         balFactor;
    DataType        key;
    struct tagAVLNode*      pLeft;
    struct tagAVLNode*      pRight;
};
typedef   tagAVLNode  AVLNode; //Doi ten kieu cac nut
//Định nghĩa kiểu AVL
typedef AVLNode     *AVLTree; // Kieu AVL
```

Cây AVL

- Thêm hay hủy một phần tử trên cây giống như thực hiện trên BST, tuy nhiên các thao tác này có thể làm cây tăng hay giảm chiều cao, khi đó phải cân bằng lại cây. Khi cân bằng lại cây phải đảm bảo tính BST.
- Việc cân bằng lại một cây thực hiện sao cho ảnh hưởng tối thiểu đến cây nhằm giảm thiểu chi phí cân bằng.
- Các thao tác đặc trưng của cây AVL:
  - Thêm một phần tử vào cây AVL.
  - Hủy một phần tử trên cây AVL.
  - Cân bằng lại một cây vừa bị mất cân bằng.

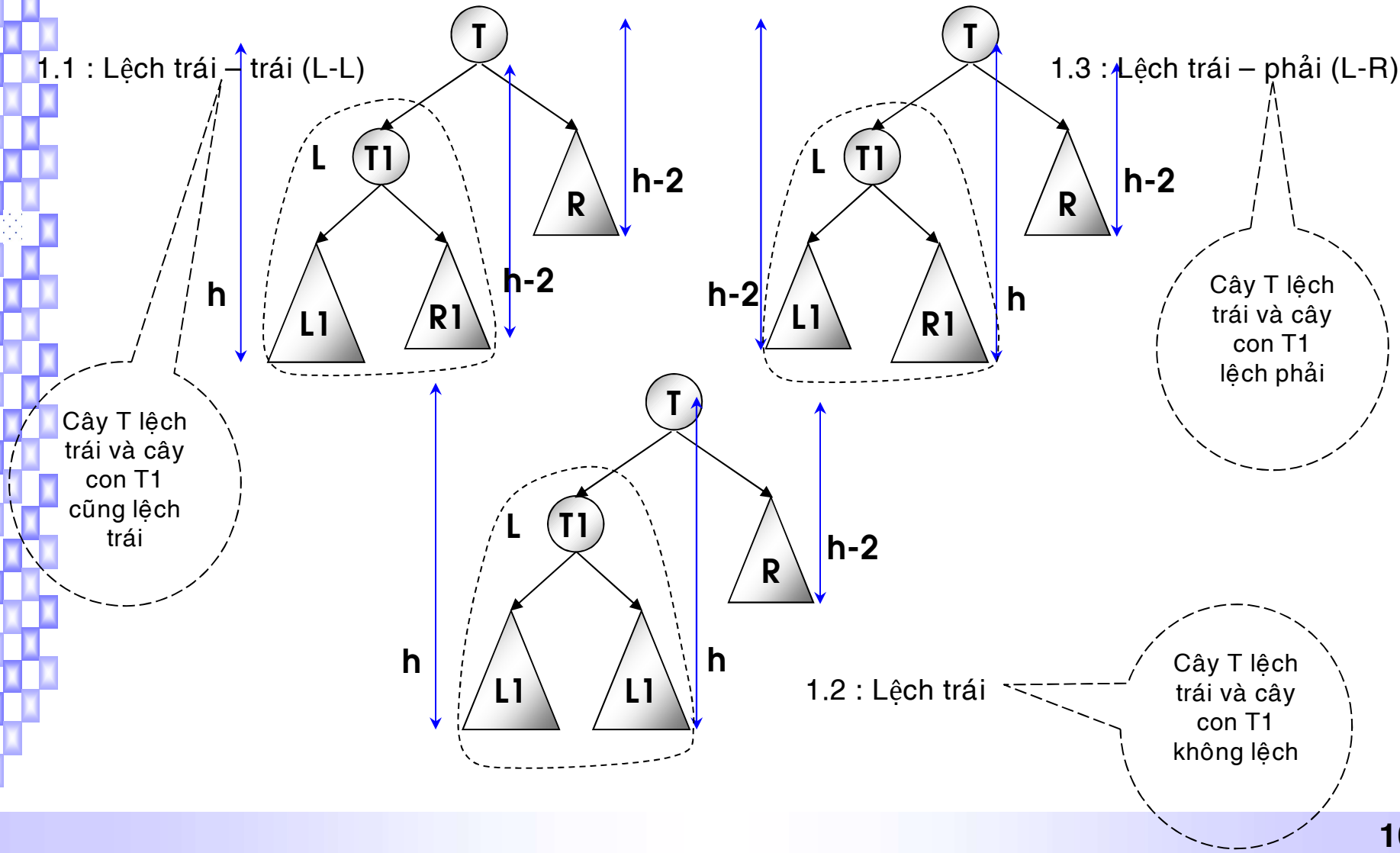
Các trường hợp mất cân bằng

- Cây AVL mất cân bằng khi độ lệch chiều cao giữa 2 cây con sẽ là 2.

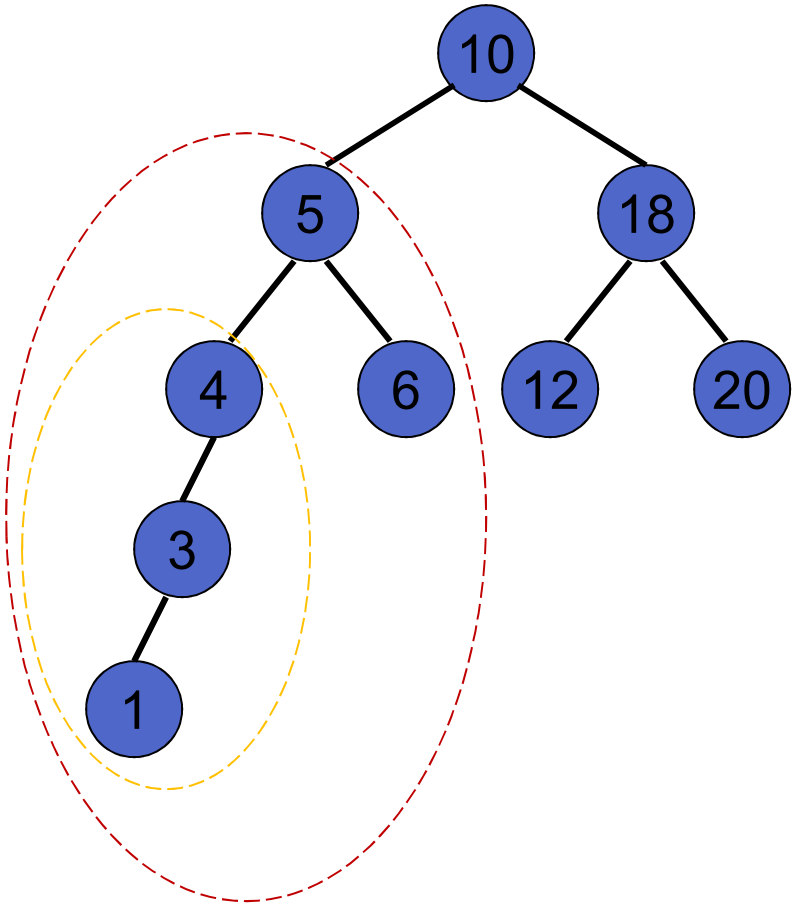
  (do thêm hay hủy một nút chỉ có thể làm cho chiều cao của cây tăng hay giảm nhiều lắm là 1)

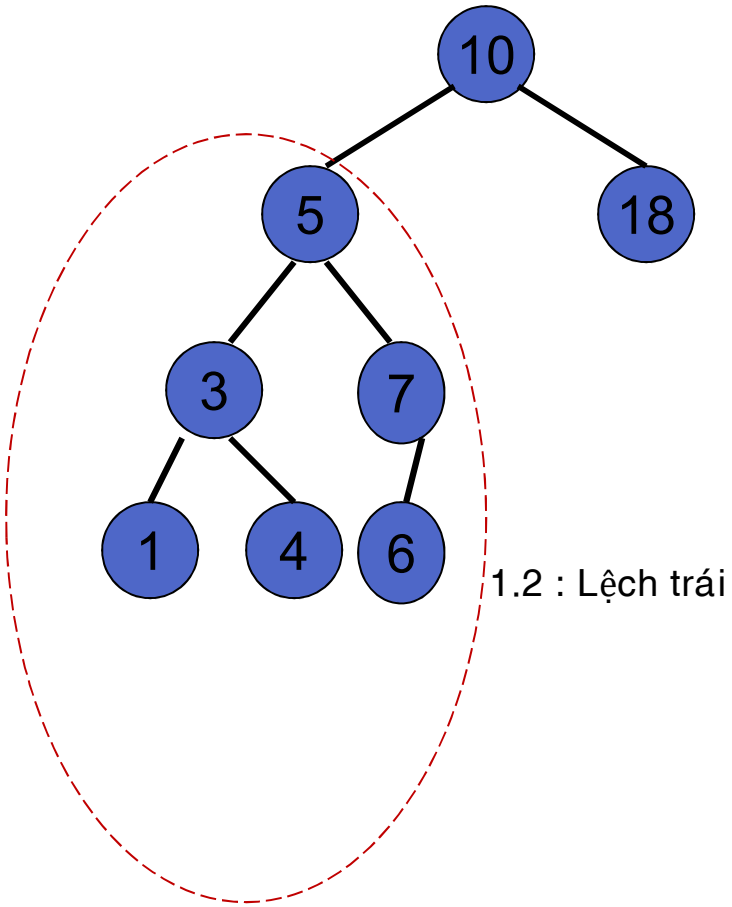
- Có 6 khả năng sau:
  - **Trường hợp 1** - Cây T lệch về bên trái : 3 khả năng
  - **Trường hợp 2** - Cây T lệch về bên phải: 3 khả năng

Trường hợp 1: cây T lệch về bên trái (có 3 khả năng)

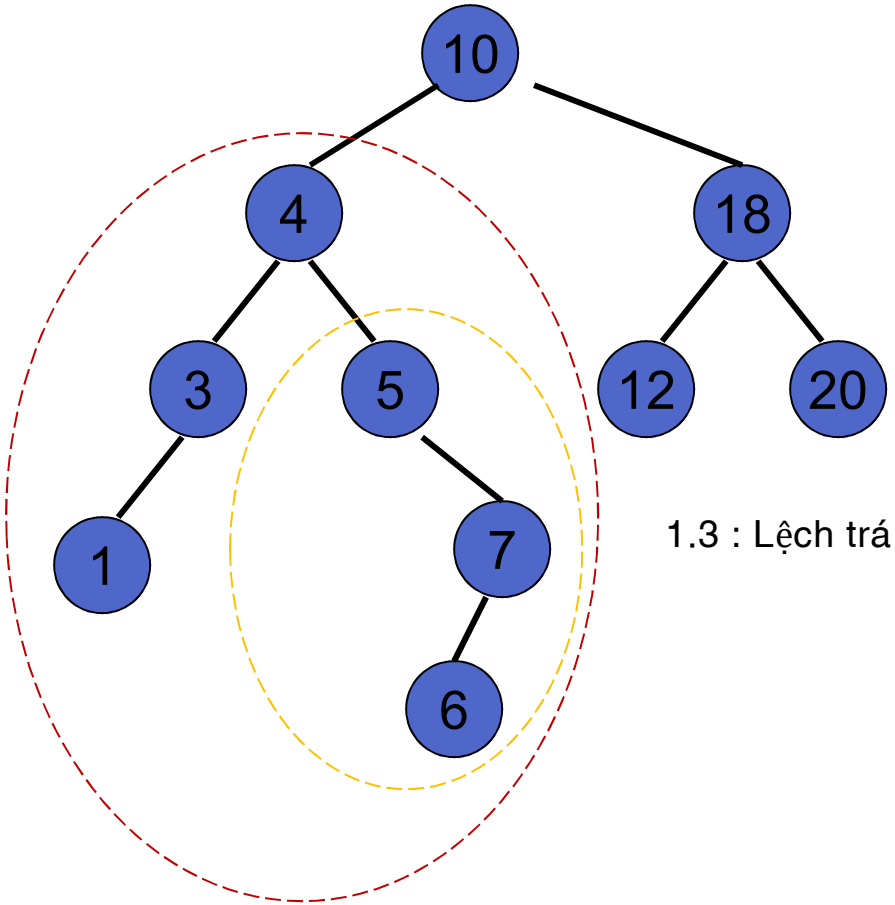


1.1 : Lệch trái – trái (L-L)





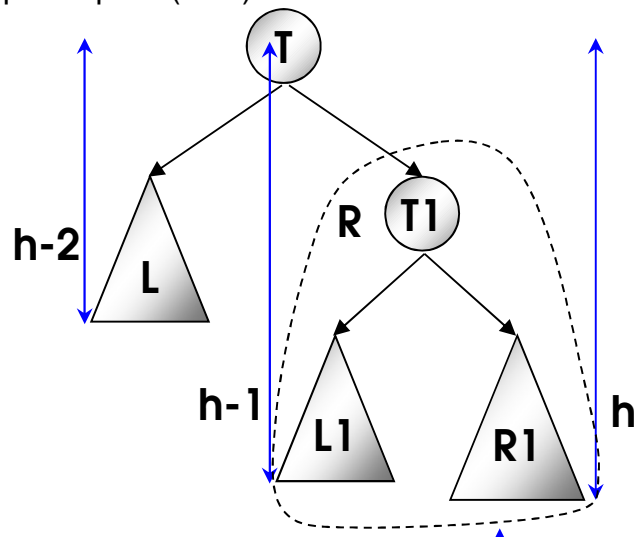
1.2 : Lệch trái



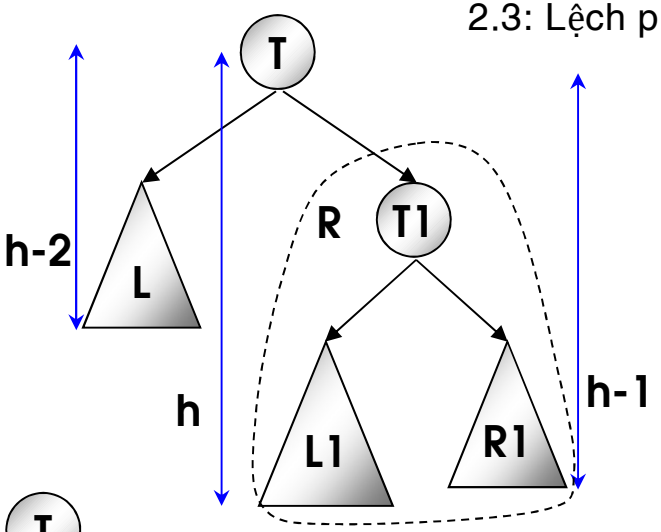
1.3 : Lệch trái – phải (L-R)

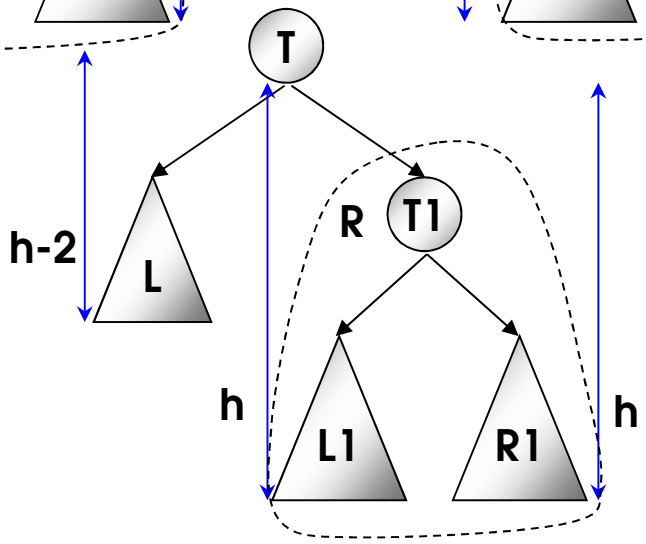
Trường hợp 2: cây T lệch về bên phải (có 3 khả năng)

2.1 :Lệch phải– phải (R-R)



2.3: Lệch phải– trái (P-R)





2.2 : Lệch phải

Các trường hợp mất cân bằng

- Các trường hợp lệch về bên phải hoàn toàn đối xứng với các trường hợp lệch về bên trái.
- Vì vậy, chỉ cần khảo sát trường hợp lệch về bên trái.
- Trong 3 trường hợp lệch về bên trái, trường hợp 1.3 lệch trái - phải là phức tạp nhất.
- Để thực hiện cân bằng cây, ta cần một thao tác gọi là quay. Có thể quay 1 lần (quay đơn), nhiều khi phải quay 2 lần (quay kép)

Phép quay

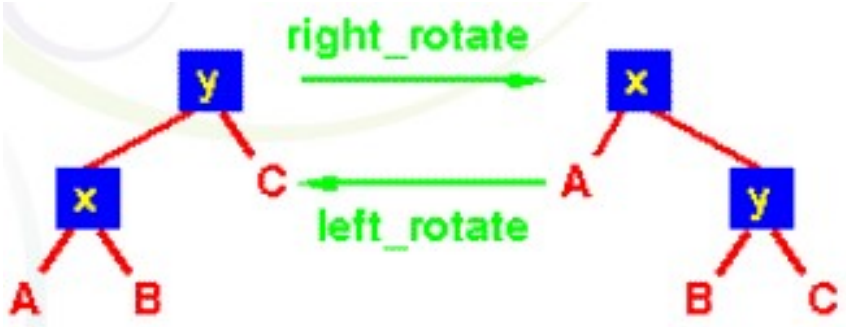
- Phép quay là cách tái sắp xếp các nút, được thiết kế làm các công việc sau:
  - Nâng một số nút lên và hạ một số nút khác xuống để giúp cân bằng cây.
  - Bảo đảm những tính chất của cây tìm kiếm nhị phân không bị vi phạm.

- Thuật ngữ quay có thể bị hiểu nhầm. Thực ra quay không có nghĩa là các nút bị quay mà là chỉ ***sự thay đổi quan hệ giữa chúng***.

Một nút (Y) được chọn làm "đỉnh" của phép quay. Nếu ta thực hiện một phép quay sang phải từ nút đỉnh này (Y), phép quay phải hoạt động như sau :

- Nút "đỉnh" (Y) này sẽ di chuyển xuống dưới và về bên phải, vào vị trí của nút con bên phải của nó (C).
- Nút con bên trái cũ của Y (là X) sẽ đi lên để chiếm lấy vị trí cũ của nó (X trở thành đỉnh – gốc) – (Y trở thành con phải của X)
- Con trái cũ của X (là A) tiếp tục là con trái của X.
- Con phải cũ của của X (là B, >X và <Y) nên làm con trái của Y.
- Con phải cũ của của Y (là C) trở thành con phải của Y

(dùng tính chất BST)



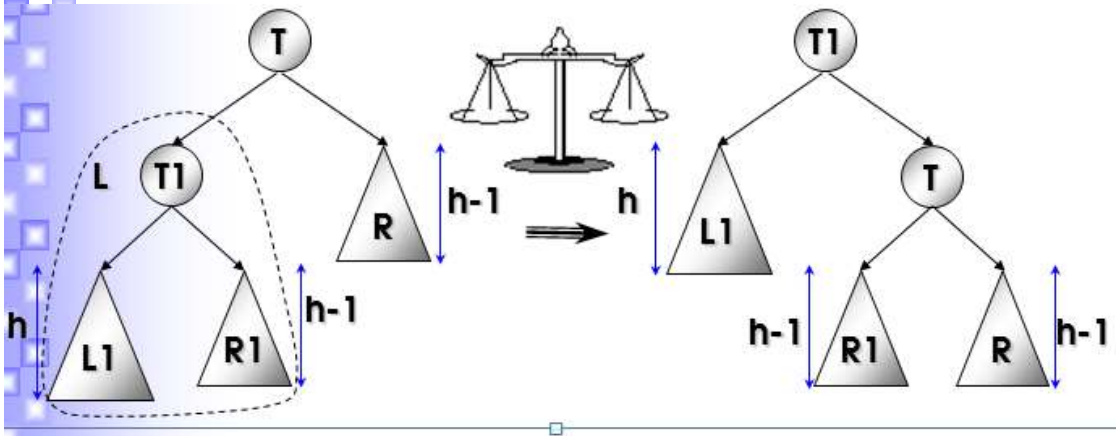
Quay trái và quay phải

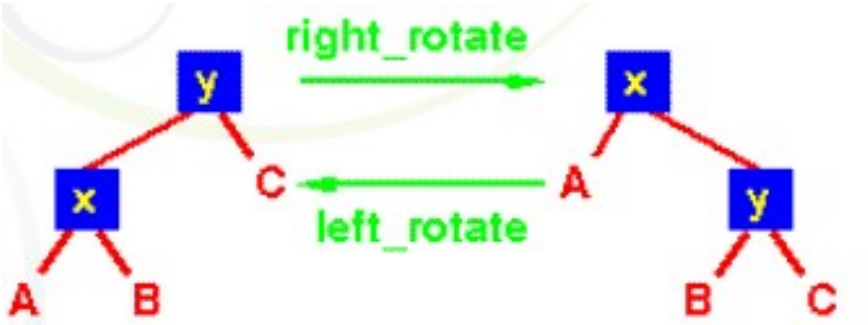
- Kết quả của 2 phép quay thứ tự duyệt cây trong phép duyệt LNR không thay đổi: **A x B y C**
- Yêu cầu về phép quay phải:
  - Nút đỉnh của quay phải (Y) bắt buộc có ***nút con trái***.

  (Nếu không chẳng có gì để quay vào điểm đỉnh.)
- Yêu cầu về phép quay trái:
  - Nút đỉnh của quay trái (X) bắt buộc có ***nút con phải***.

Cân bằng lại cây AVL

- T/h 1.1: cây T1 lệch về bên trái. Ta thực hiện phép quay đơn Left-Left (ta thực hiện quay phải).





- Thuật toán quay đơn Left - Left

B1:

gốc T;

T1 = T->pLeft;

T->pLeft = T1->pRight;

T1->pRight = T;

B2: //đĐặt lại chỉ số cân bằng

Nếu T1->balFactor = LH thì:

T->balFactor = EH;

T1->balFactor = EH;

break;

Nếu T1->balFactor = EH thì:

T->balFactor = LH;

T1->balFactor = RH;

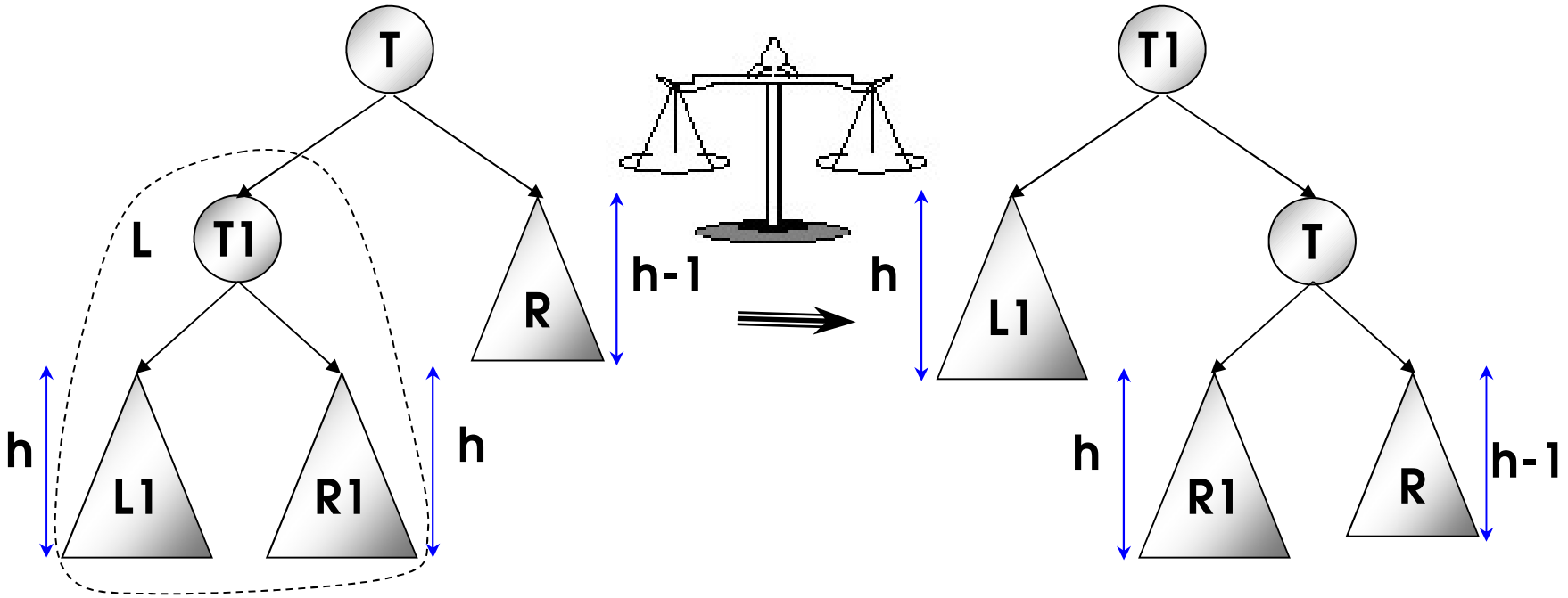
break;

B3: //trỏ đến gốc mới

T = T1;

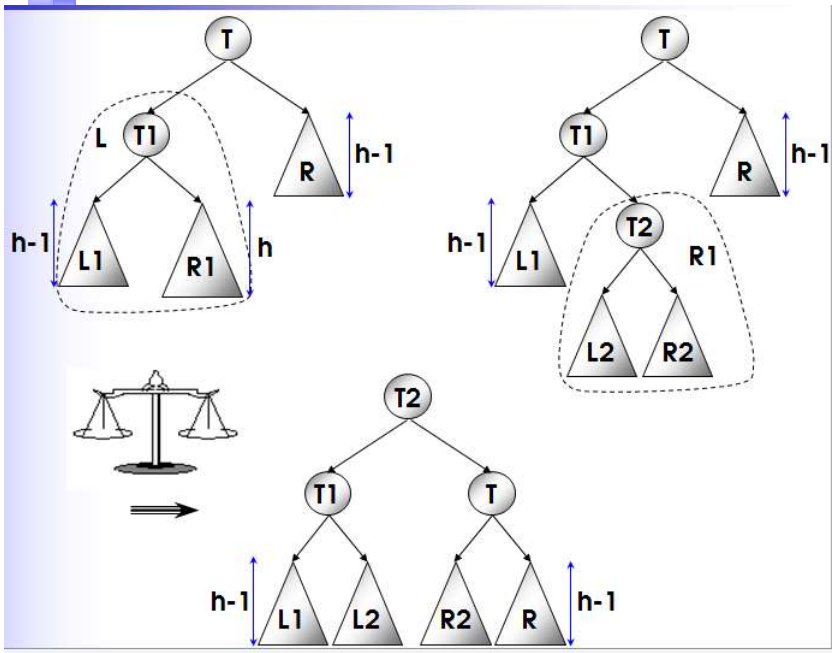
Cân bằng lại cây AVL

- T/h 1.2: cây T1 không lệch. Ta thực hiện phép quay đơn Left-Left



Cân bằng lại cây AVL

- T/h 1.3: cây T1 lệch về bên phải.
- Do T1 lệch về bên phải ta không thể áp dụng phép quay đơn đã áp dụng trong 2 trường hợp trên vì khi đó cây T sẽ chuyển từ trạng thái mất cân bằng do lệch trái thành mất cân bằng do lệch phải $\Rightarrow$ cần áp dụng cách khác.
- Ta thực hiện phép quay kép Left-Right



- Thuật toán quay kép Left – Right

B1:

gốc T;

T1 = T->pLeft; T2 = T1->pRight; T->pLeft = T2->pRight;

T2->pRight = T;T1->pRight = T2->pLeft;T2->pLeft = T1;

B2: //đĐặt lại chỉ số cân bằng

Nếu T2->balFactor = LH thì:

T->balFactor = RH;

T1->balFactor = EH;

Ngược lại,

Nếu T2->balFactor = EH thì:

T->balFactor = EH;

T1->balFactor = EH;

Ngược lại,

Nếu T2->balFactor = RH thì:

T->balFactor = EH;

T1->balFactor = LH;

B3:

T2->balFactor = EH;

T = T2;

Cân bằng lại cây AVL

Quay đơn Left-Left

```
void rotateLL(AVLTree &T) //quay đơn Left-Left
{
   AVLNode*  T1 = T->pLeft;
   T->pLeft       = T1->pRight;
   T1->pRight     = T;
   switch(T1->balFactor) {
   case LH: T->balFactor  = EH;
            T1->balFactor = EH;
            break;
   case EH: T->balFactor  = LH;
            T1->balFactor = RH;
            break;
   }
   T= T1;
}
```

Cân bằng lại cây AVL

Quay đơn Right-Right

```
void rotateRR(AVLTree &T)//quay đơn Right-Right
{
   AVLNode*        T1 = T->pRight;
   T->pRight       = T1->pLeft;
   T1->pLeft       = T;
   switch(T1->balFactor) {
   case RH: T->balFactor = EH;
            T1->balFactor= EH;
            break;
   case EH: T->balFactor = RH;
            T1->balFactor= LH;
            break;
   }
   T= T1;
}
```

Cân bằng lại cây AVL

Quay ́kp Left-Right

```
void rotateLR(AVLTree &T)//quay kép Left-Right
{  AVLNode*   T1 = T->pLeft;
   AVLNode*   T2 = T1->pRight;
   T->pLeft   = T2->pRight;
   T2->pRight = T;
   T1->pRight = T2->pLeft;
   T2->pLeft  = T1;
   switch(T2->balFactor)    {
     case LH: T->balFactor  = RH; T1->balFactor = EH; break;
     case EH: T->balFactor  = EH; T1->balFactor = EH; break;
     case RH: T->balFactor  = EH; T1->balFactor = LH; break;
   }
   T2->balFactor = EH;
   T = T2;
}
```

Cân bằng lại cây AVL

Quay kép Right-Left

```
void rotateRL(AVLTree &T)//quay kép Right-Left
{  AVLNode*   T1 = T->pRight;
   AVLNode*   T2 = T1->pLeft;
   T->pRight  = T2->pLeft;
   T2->pLeft  = T;
   T1->pLeft  = T2->pRight;
   T2->pRight = T1;
   switch(T2->balFactor)    {
     case RH: T->balFactor = LH; T1->balFactor = EH; break;
     case EH: T->balFactor = EH; T1->balFactor = EH; break;
     case LH: T->balFactor = EH; T1->balFactor = RH; break;
   }
   T2->balFactor     = EH;
   T    = T2;
}
```

Cân bằng lại cây AVL

Cân bằng khi cây bị lệch trái

```
int balanceLeft(AVLTree &T)//Cân bằng khi cây bị lệch về bên trái
{
   AVLNode*   T1 = T->pLeft;

   switch(T1->balFactor)                {
   case LH:   rotateLL(T); return 2;
   case EH:   rotateLL(T); return 1;
   case RH:   rotateLR(T); return 2;
   }
   return 0;
}
```

Cân bằng lại cây AVL

Cân bằng khi cây bị lệch phải

```
int balanceRight(AVLTree &T)//Cân bằng khi cây bị lệch về bên
   phải
{
   AVLNode*   T1 = T->pRight;

   switch(T1->balFactor)                {
   case LH:   rotateRL(T); return 2;
   case EH:   rotateRR(T); return 1;
   case RH:   rotateRR(T); return 2;
   }
   return 0;
}
```

Thêm một phần tử trên cây AVL

- Việc thêm một phần tử vào cây AVL diễn ra tương tự như trên BST.
- Sau khi thêm xong, nếu chiều cao của cây thay đổi, từ vị trí thêm vào, ta phải lần ngược lên gốc để kiểm tra xem có nút nào bị mất cân bằng không. Nếu có, ta phải cân bằng lại ở nút này.
- Việc cân bằng lại **chỉ cần thực hiện 1 lần tại nơi mất cân bằng**.
- Hàm `insertNode` trả về giá trị –1, 0, 1 khi không đủ bộ nhớ, gặp nút cũ hay thành công. Nếu sau khi thêm, chiều cao cây bị tăng, giá trị 2 sẽ được trả về

```
int insertNode(AVLTree &T, DataType X)
```

Thêm một phần tử trên cây AVL

insertNode (1/3)

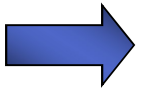
```
int insertNode(AVLTree &T, DataType X)
{  int res;
   if (T)
   {    if (T->key == X) return 0; //đã có
        if (T->key > X)
        {    res       = insertNode(T->pLeft, X);
             if(res < 2) return res;
             switch(T->balFactor)
             {  case RH: T->balFactor = EH;  return 1;
                case EH: T->balFactor = LH;  return 2;
                case LH: balanceLeft(T);     return 1;
             }
        }
.....................................................
   .
}
```

insertNode2

Thêm một phần tử trên cây AVL

insertNode (2/3)

```
int insertNode(AVLTree &T, DataType X)
{
.....................................................
        else // T->key < X
        {   res = insertNode(T-> pRight, X);
            if(res < 2) return res;
            switch(T->balFactor)
            {   case LH: T->balFactor  = EH; return 1;
                case EH: T->balFactor  = RH; return 2;
                case RH: balanceRight(T);     return 1;
                }
        }
.....................................................
}
```



insertNode3

Thêm một phần tử trên cây AVL

insertNode (3/3)

```
int insertNode(AVLTree &T, DataType X)
{
.....................................................
   .
   T    = new TNode;
   if(T == NULL) return -1; //thiếu bộ nhớ
   T->key = X;
   T->balFactor      = EH;
   T->pLeft = T->pRight = NULL;
   return 2; // thành công, chiều cao tăng
}
```

Hủy một phần tử trên cây AVL

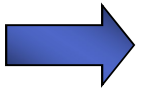
- Cũng giống như thao tác thêm một nút, việc hủy một phần tử X ra khỏi cây AVL thực hiện giống như trên BST.
- Sau khi hủy, nếu tính cân bằng của cây bị vi phạm ta sẽ thực hiện việc cân bằng lại.
- Tuy nhiên việc cân bằng lại trong thao tác hủy sẽ phức tạp hơn nhiều do có thể xảy ra phản ứng dây chuyền.
- Hàm **delNode** trả về giá trị 1, 0 khi hủy thành công hoặc không có X trong cây. Nếu sau khi hủy, chiều cao cây bị giảm, giá trị 2 sẽ được trả về:

```
int delNode(AVLTree &T, DataType X)
```

Hủy một phần tử trên cây AVL

delNode (1/3)

```
int delNode(AVLTree &T, DataType X)
{  int res;
   if(T==NULL)          return 0;
   if(T->key > X)
   {   res = delNode (T->pLeft, X);
       if(res < 2)     return res;
       switch(T->balFactor)
       {   case LH: T->balFactor = EH; return 2;
           case EH: T->balFactor = RH; return 1;
           case RH: return balanceRight(T);
       }
   } // if(T->key > X)
.....................................................
}
```

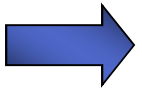


delNode2

Hủy một phần tử trên cây AVL

delNode (2/3)

```
int delNode(AVLTree &T, DataType X)
{
.....................................................
   if(T->key < X)
   {  res     = delNode (T->pRight, X);
      if(res < 2)    return res;
      switch(T->balFactor)
      {  case RH: T->balFactor = EH; return 2;
         case EH: T->balFactor = LH; return 1;
         case LH: return balanceLeft(T);
      }
   } // if(T->key < X)
.....................................................
}
```



delNode3

Hủy một phần tử trên cây AVL

delNode (3/3)

```
int delNode(AVLTree &T, DataType X)
{.....................................................
   else //T->key == X
   {  AVLNode*   p = T;
      if(T->pLeft == NULL)        {  T = T->pRight; res = 2; }
      else if(T->pRight == NULL) {  T = T->pLeft;  res = 2; }
           else //T có đủ cả 2 con
           {  res = searchStandFor(p,T->pRight);
              if(res < 2) return res;
              switch(T->balFactor)
              {  case RH: T->balFactor = EH; return 2;
                 case EH: T->balFactor = LH; return 1;
                 case LH: return balanceLeft(T);
              }
           }
      delete p;  return res;
   }
}
```

Hủy một phần tử trên cây AVL

Tìm phần tử thế mạng searchStandFor

```
int searchStandFor(AVLTree &p, AVLTree &q)
//Tìm phần tử thế mạng
{  int res;
   if(q->pLeft)
   {  res     = searchStandFor(p, q->pLeft);
      if(res < 2)    return res;
      switch(q->balFactor)
      {  case LH: q->balFactor = EH; return 2;
         case EH: q->balFactor = RH; return 1;
         case RH: return balanceRight(T);
      }
   }  else
   {  p->key = q->key; p = q; q = q->pRight; return
   2;
   }
}
```

Cây AVL

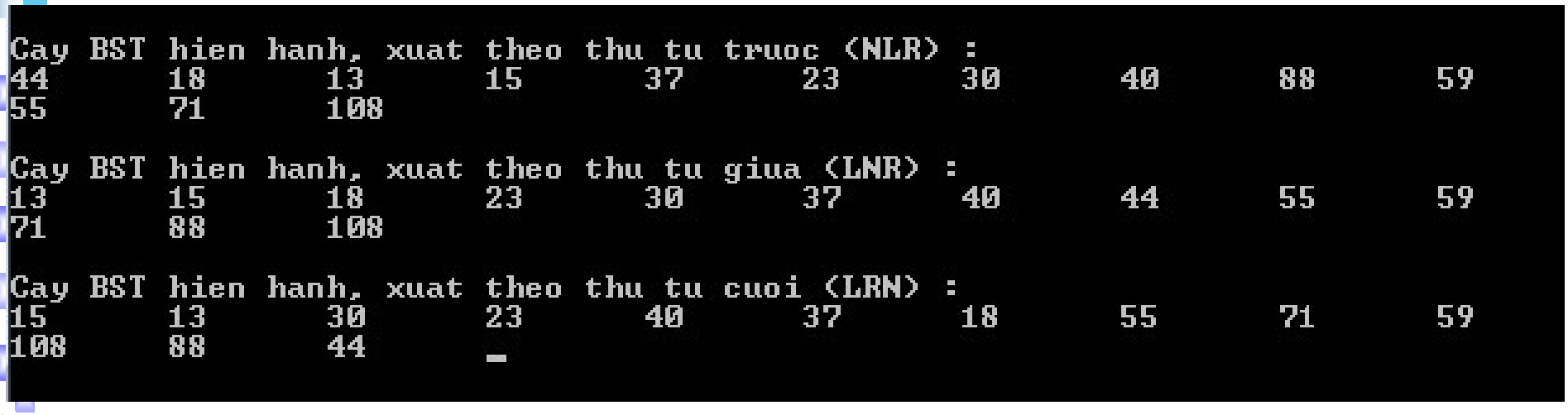
Nhận xét:

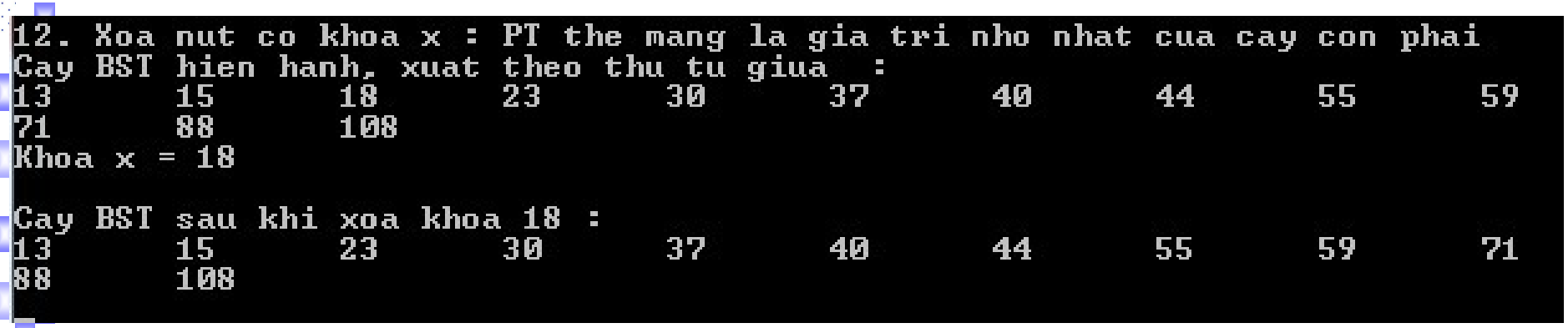
- Thao tác thêm một nút có độ phức tạp O(1).
- Thao tác hủy một nút có độ phức tạp O(h).
- Với cây cân bằng trung bình 2 lần thêm vào cây thì cần một lần cân bằng lại; 5 lần hủy thì cần một lần cân bằng lại.

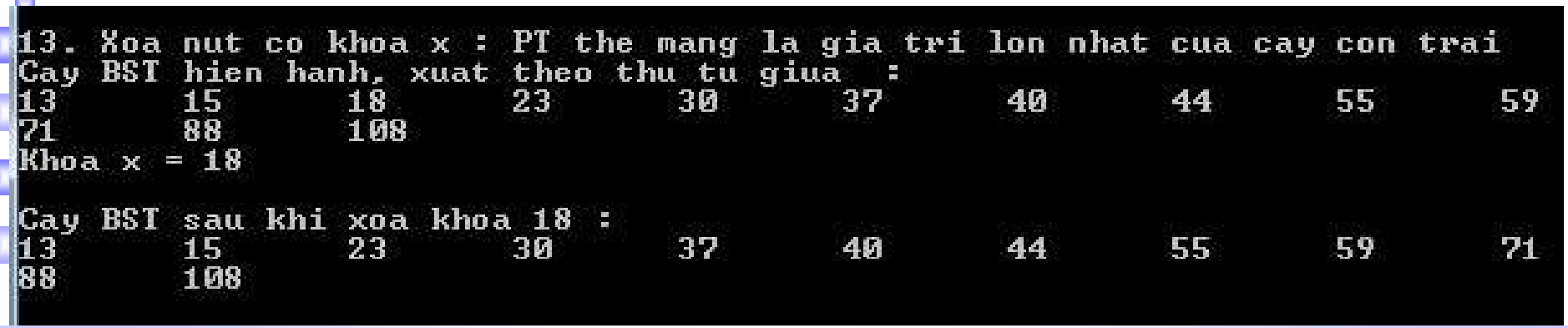
Cây AVL

Nhận xét:

- Việc huỷ 1 nút có thể phải cân bằng dây chuyền các nút từ gốc cho đên phần tử bị huỷ trong khi thêm vào chỉ cần 1 lần cân bằng cục bộ.
- Độ dài đường tìm kiếm trung bình trong cây cân bằng gần bằng cây cân bằng hoàn toàn $\log_2 n$, nhưng việc cân bằng lại đơn giản hơn nhiều.
- Một cây cân bằng không bao giờ cao hơn 45% cây cân bằng hoàn toàn tương ứng dù số nút trên cây là bao nhiêu.


Cay BST hien hanh, xuat theo thu tu truoc (NLR) :
44 18 13 15 37 23 30 40 88 59
55 71 108
Cay BST hien hanh, xuat theo thu tu giua (LNR) :
13 15 18 23 30 37 40 44 55 59
71 88 108
Cay BST hien hanh, xuat theo thu tu cuoi (LRN) :
15 13 30 23 40 37 18 55 71 59
108 88 44


12. Xoa nut co khoa x : PT the mang la gia tri nho nhat cua cay con phai
Cay BST hien hanh, xuat theo thu tu giua :
13 15 18 23 30 37 40 44 55 59
71 88 108
Khoa x = 18
Cay BST sau khi xoa khoa 18 :
13 15 23 30 37 40 44 55 59 71
88 108


13. Xoa nut co khoa x : PT the mang la gia tri lon nhat cua cay con trai
Cay BST hien hanh, xuat theo thu tu giua :
13 15 18 23 30 37 40 44 55 59
71 88 108
Khoa x = 18
Cay BST sau khi xoa khoa 18 :
13 15 23 30 37 40 44 55 59 71
88 108