

Phần 2 : Cây nhị phân tìm kiếm (Binary search tree - BST)

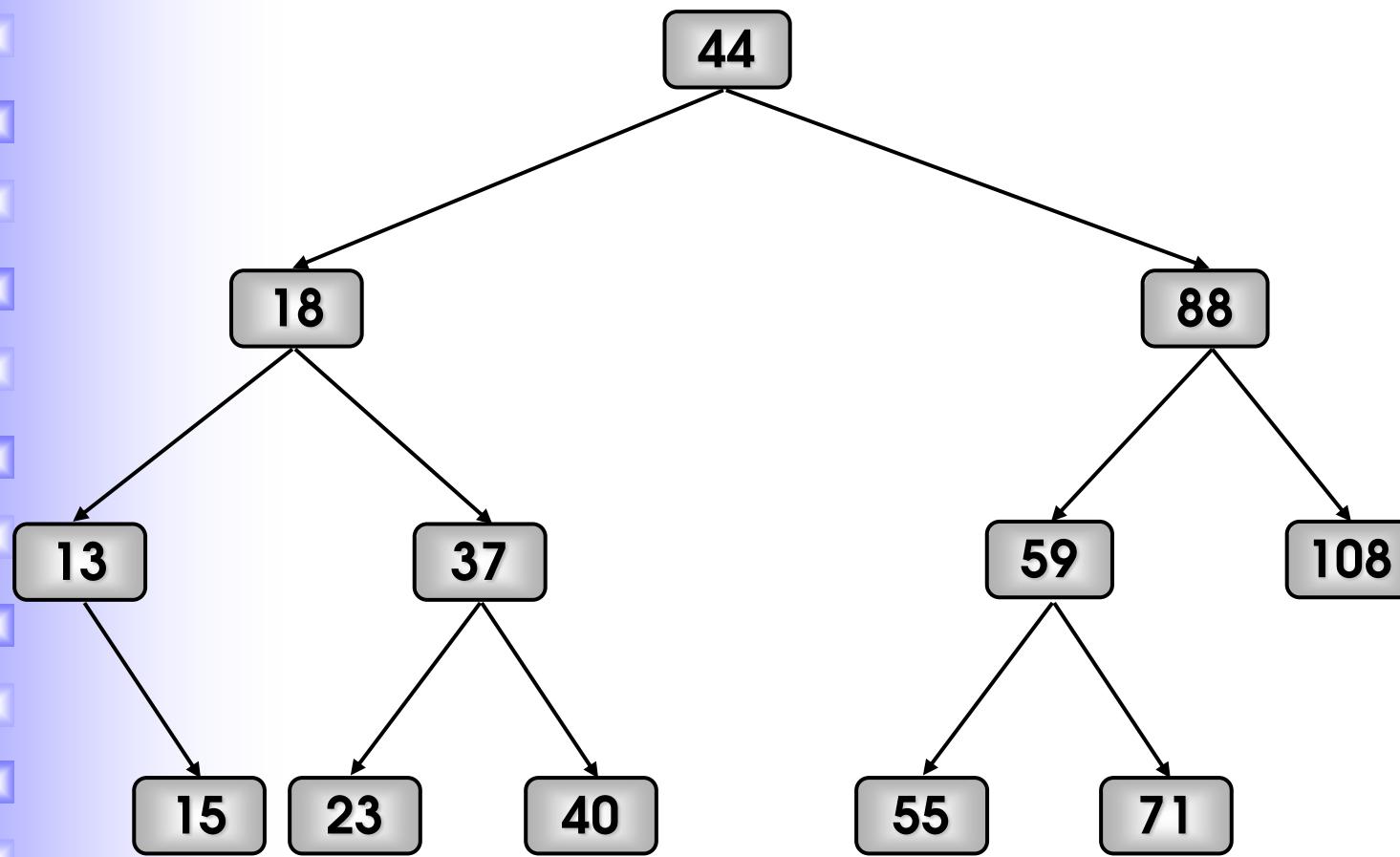
Định nghĩa và ví dụ cây nhị phân tìm kiếm (BST)

- **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 đối 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 trữ 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

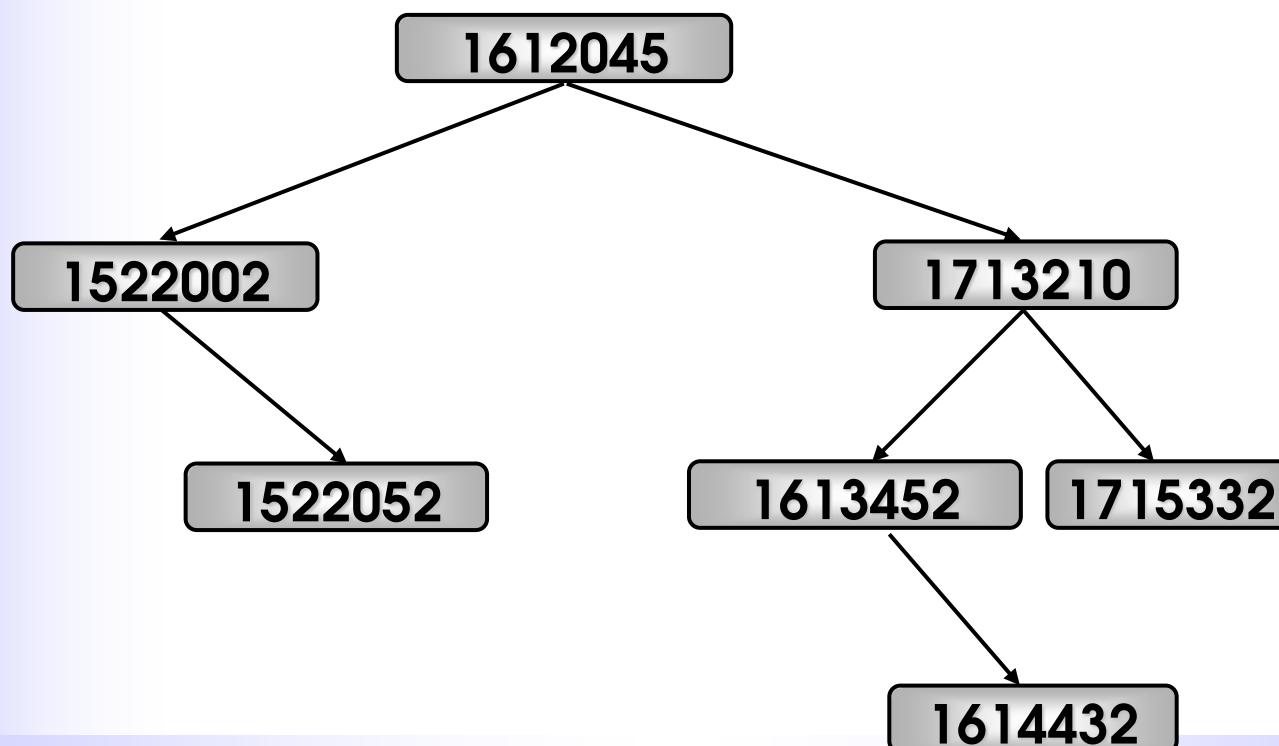
Ví dụ:

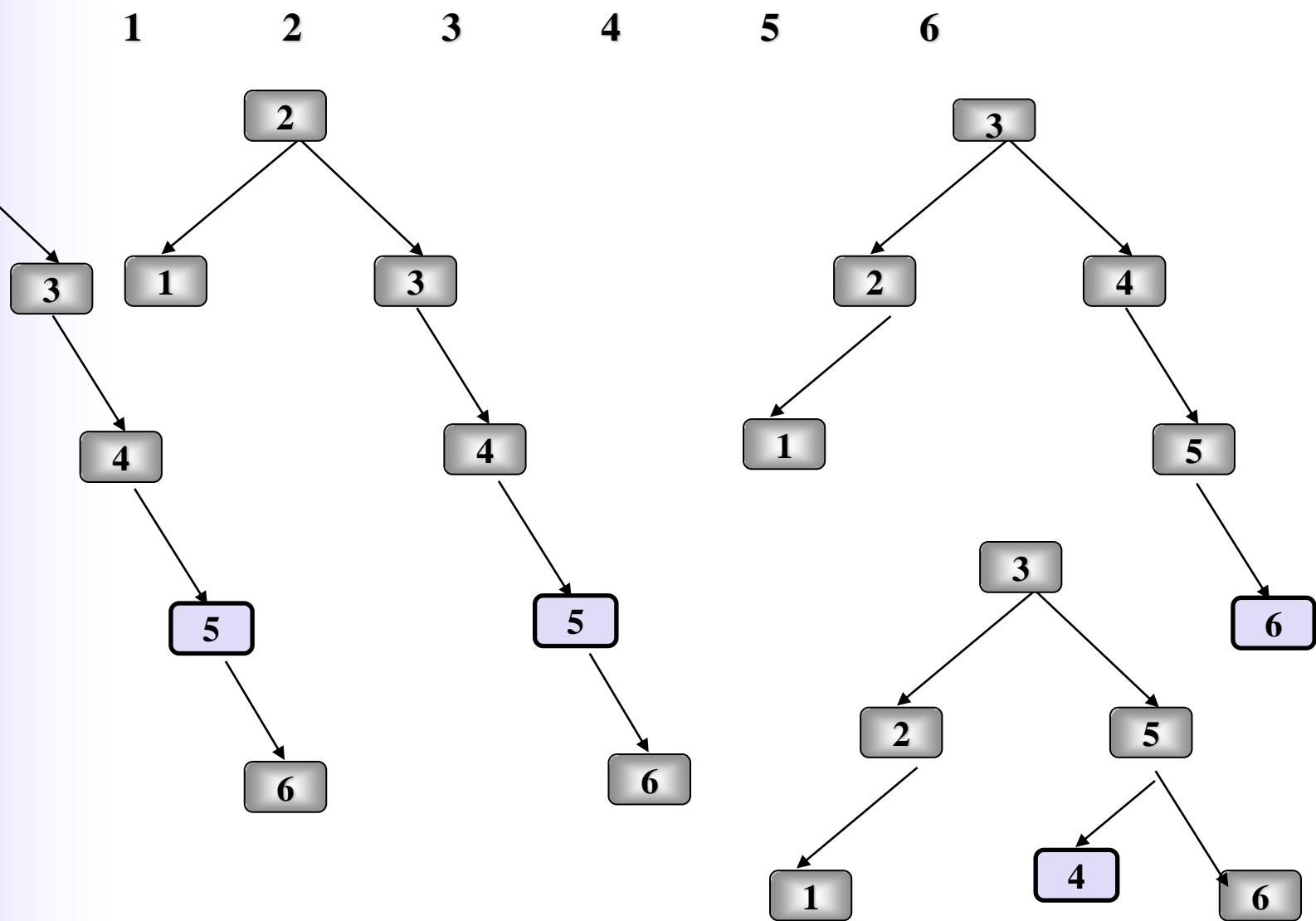


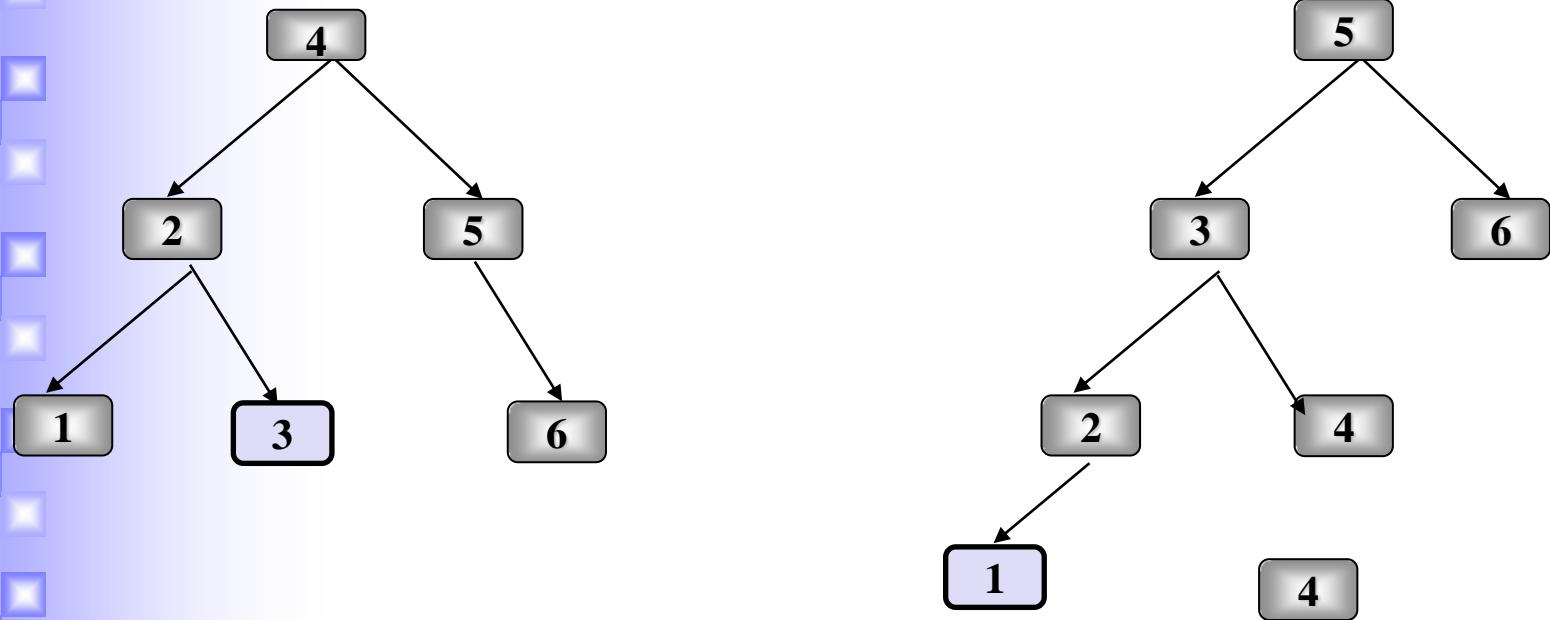
Xem bảng điểm môn học :

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 trữ trên cây BST như sau (Khi dùng khóa là trường mã sinh viên) :







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;

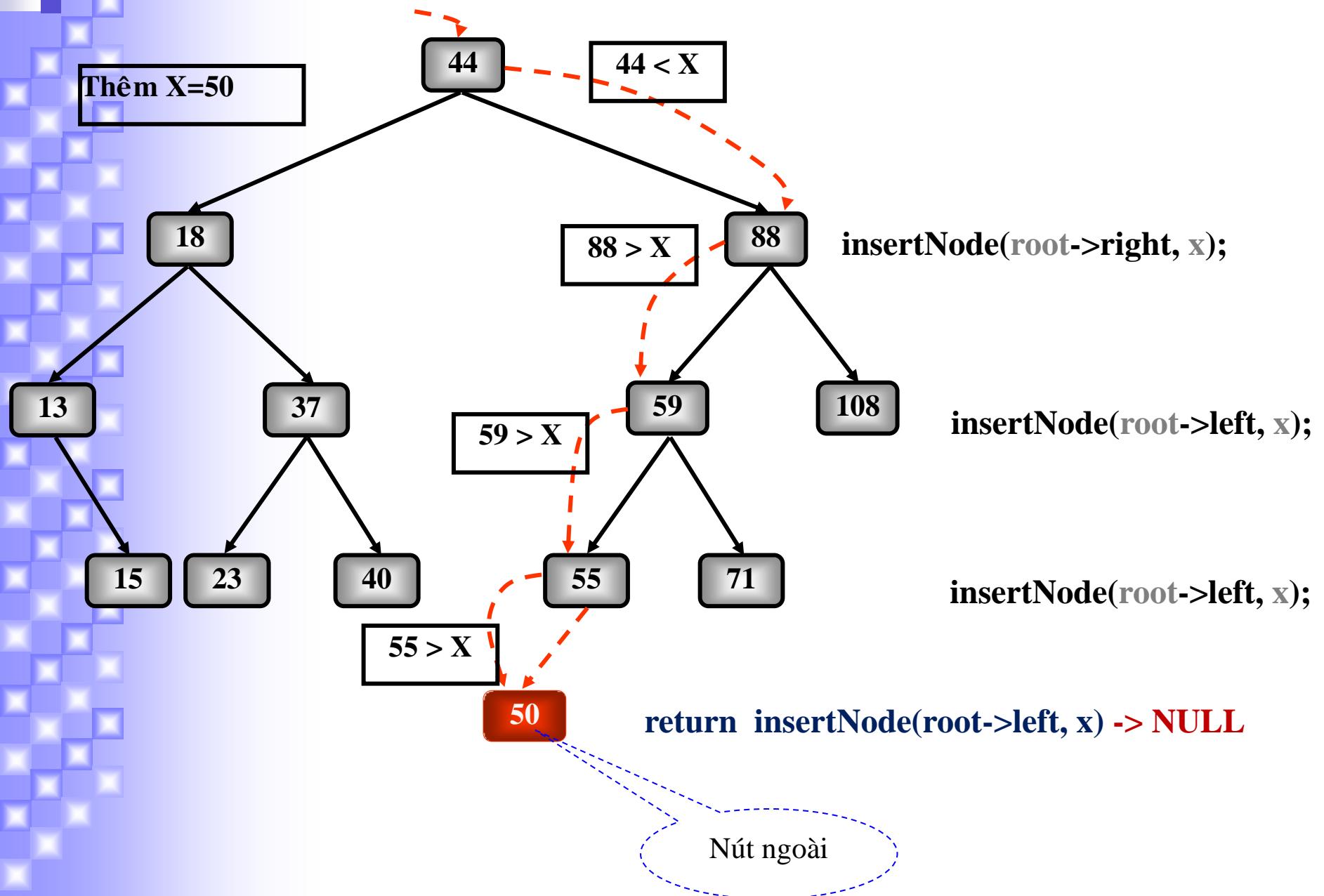
Output : -1; khong thanh cong - khong du bo nho

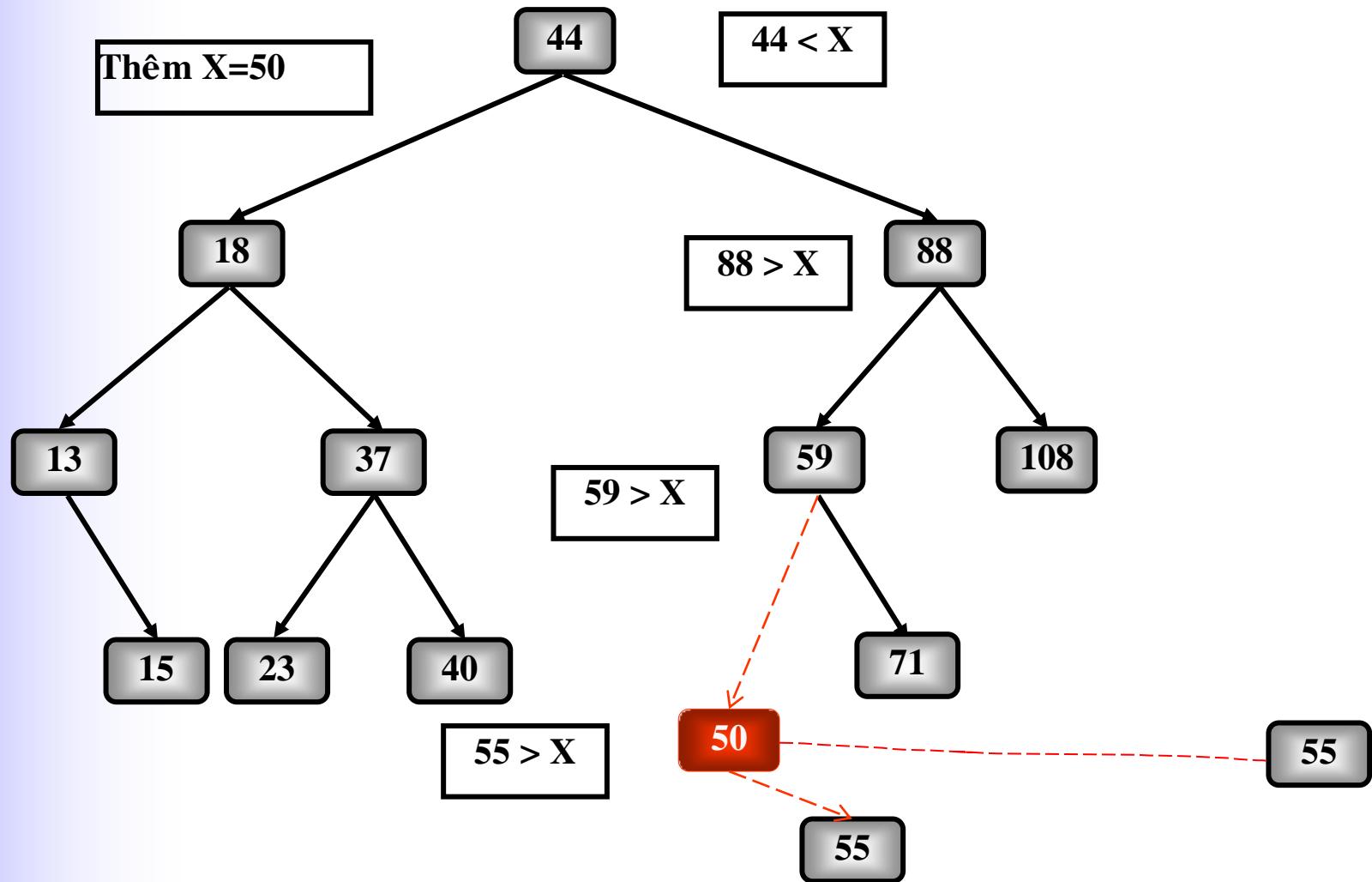
0; khong thanh cong - x da co san trong cay

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
}
```





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<<"Nhập n = "; cin>>n;
    for(i = 0; i < n; i++)
    {
        do
        {
            cout<<"Nhập giá trị: "; cin>>x;
            kq = insertNode(root,x)
            if(kq == -1)
                return; //không đủ nhớ
        } while (kq == 0); //x chưa san thi nhập lại
    }
}
```

- 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:

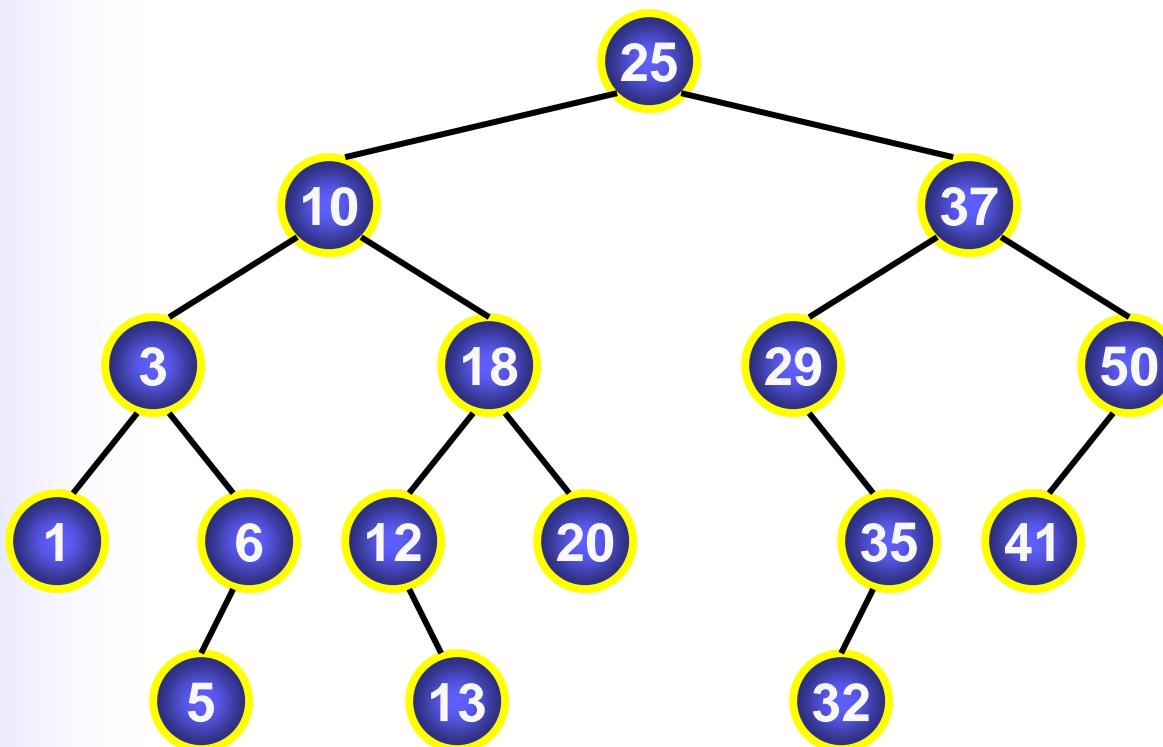
Input : filename

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);
```

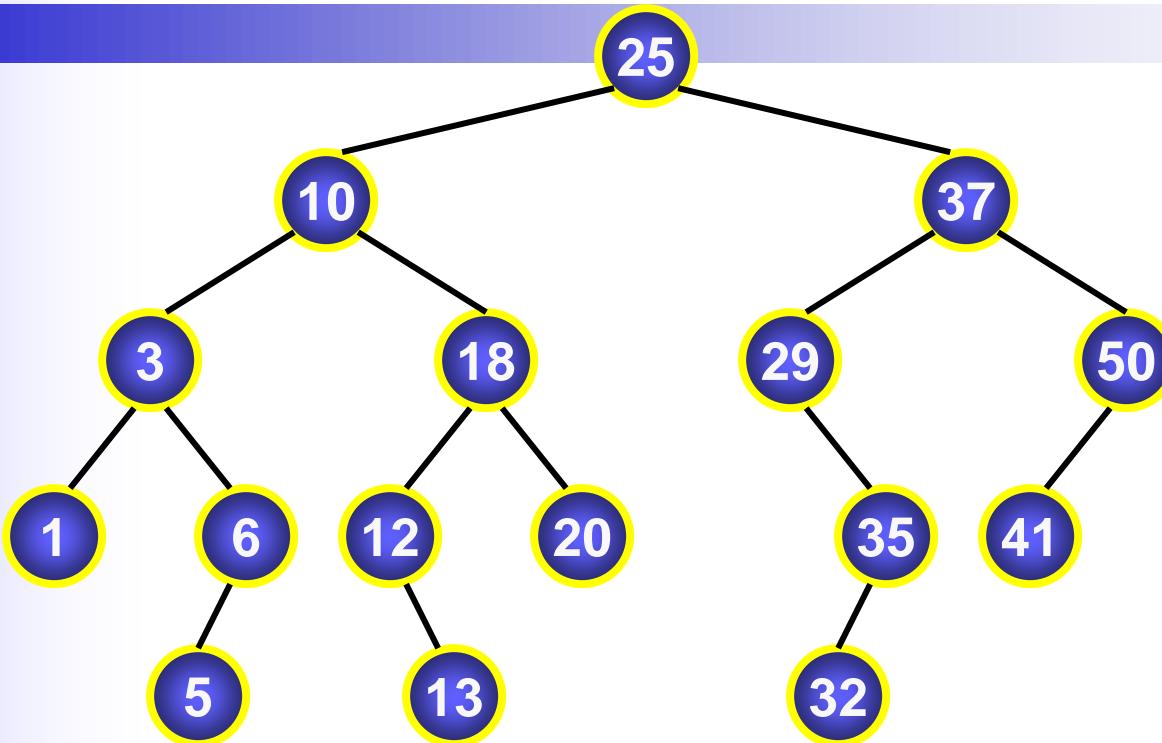
```
PosOrder(root->right);
```

```
//<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
----	----	----	----	----	----	---	---	---	---	----	----	----	----	----	----

Gay 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				

Gay 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				

Gay 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;

}

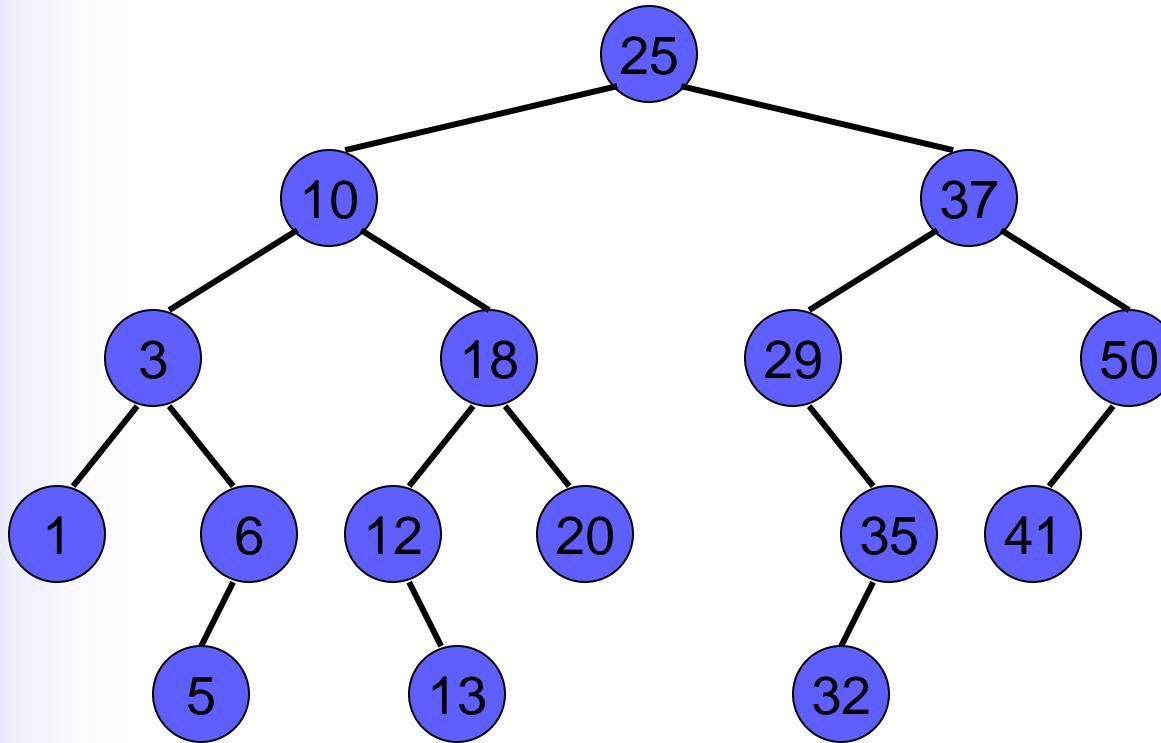
Tìm một phần tử x trong cây (không đệ quy)

- 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;  
}
```

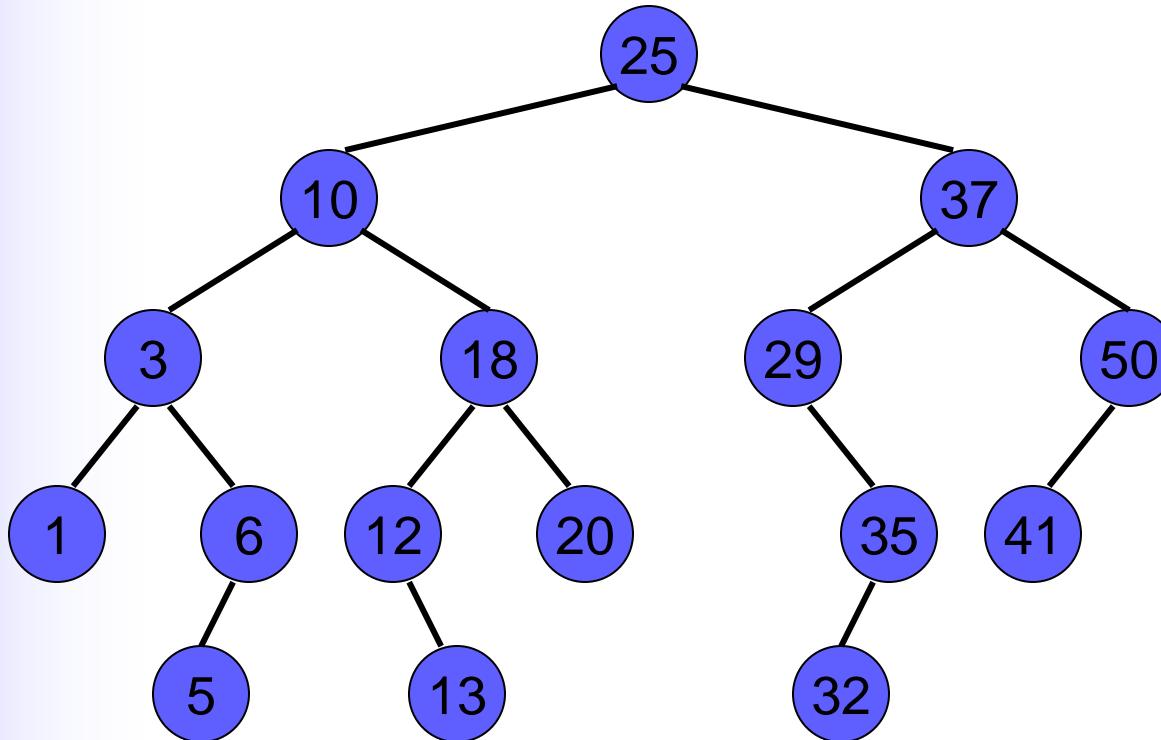
Tìm một phần tử $x=13$ trong cây



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

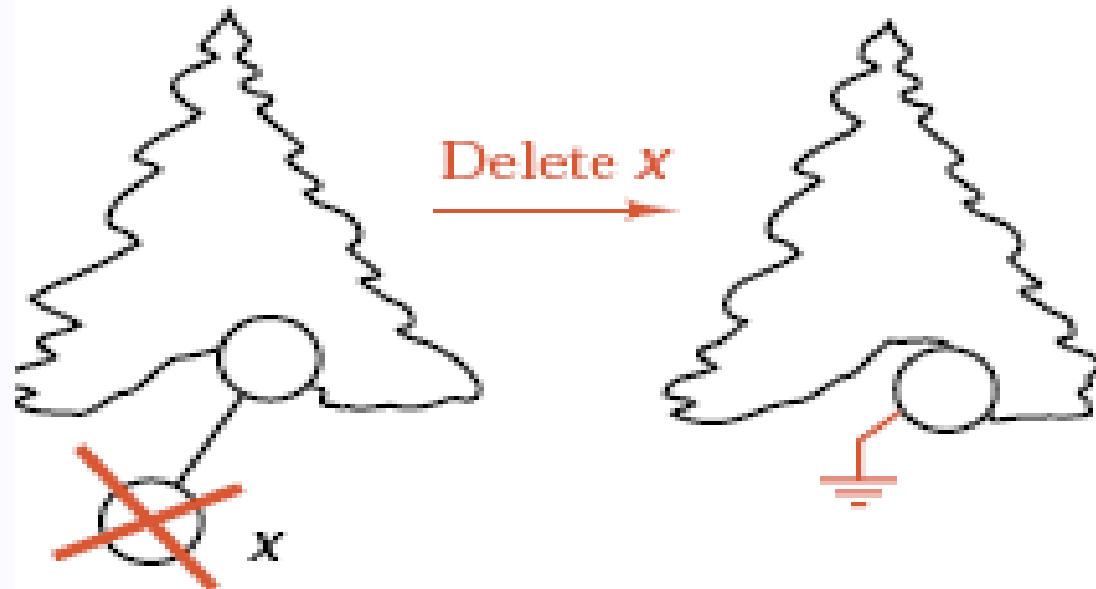
Nhận xét:

- 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

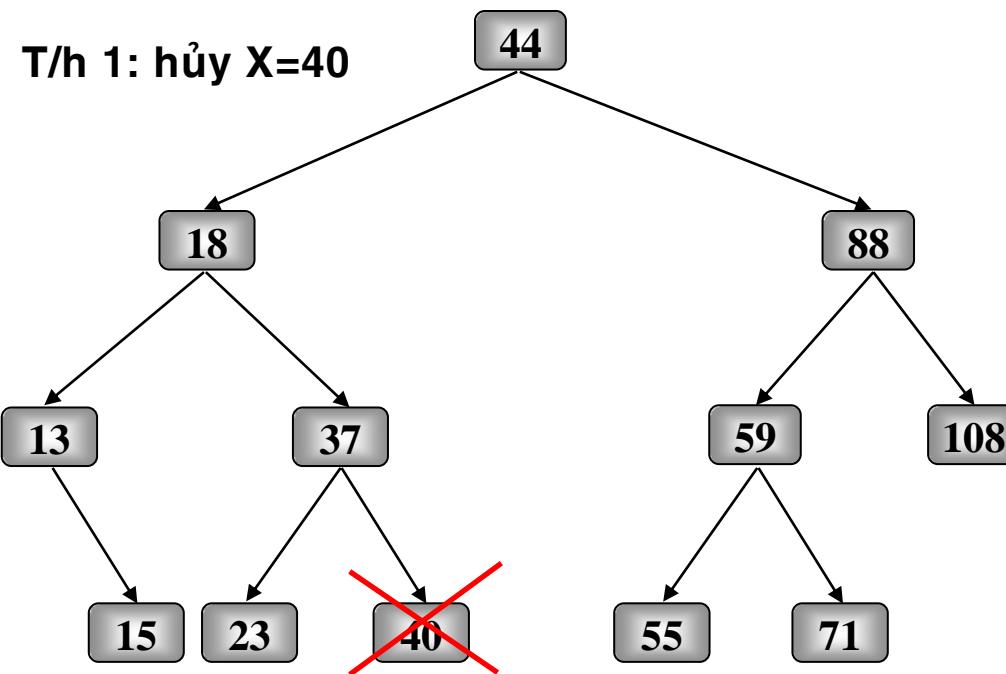
- Việc hủy một phần tử X ra khỏi cây phải bảo đảm điều kiện ràng buộc của BST.
- Có 3 trường hợp khi hủy nút X có thể xảy ra:
 - X là nút lá (không con)
 - X chỉ có 1 con (trái hoặc phải).
 - X có đủ cả 2 con

Trường hợp 1: X là nút lá.

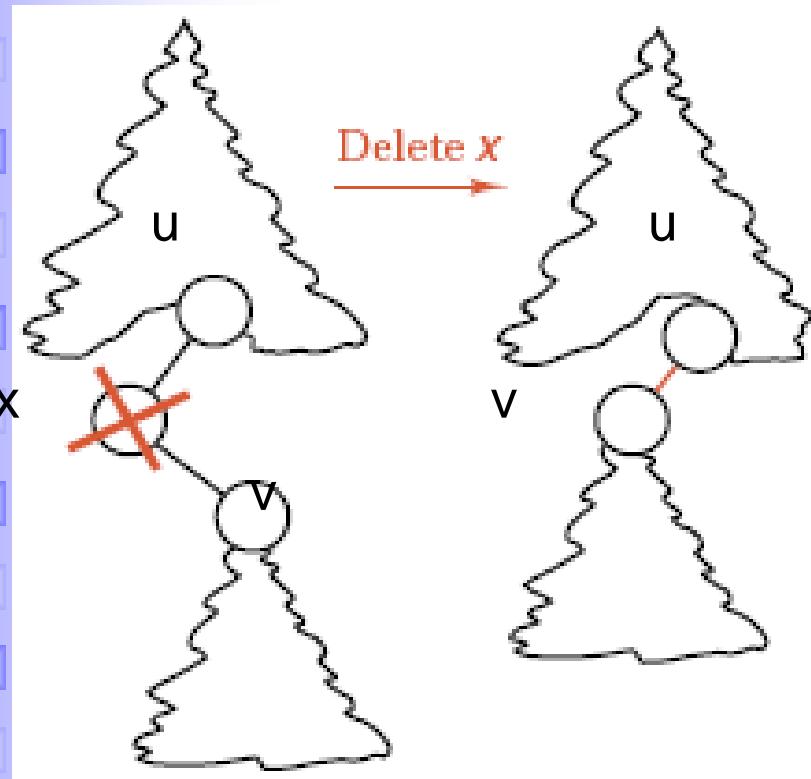


1. Xóa node này
2. Gán liên kết từ cha của nó thành rỗng

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

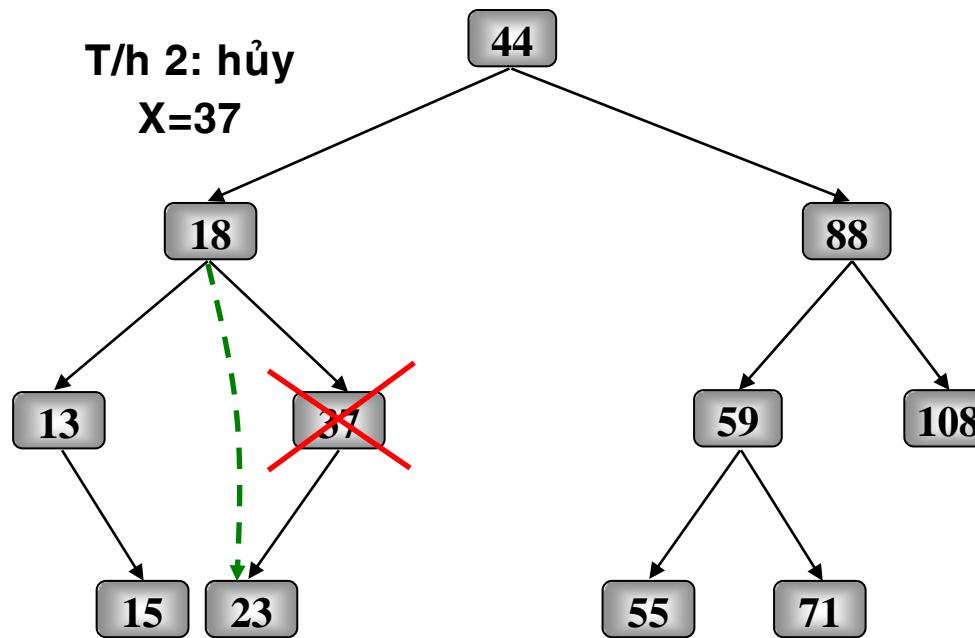


Trường hợp 2: X chỉ có 1 con (trái hoặc phải)



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 mốc nối cha của X là 18 với con duy nhất của nó là 23



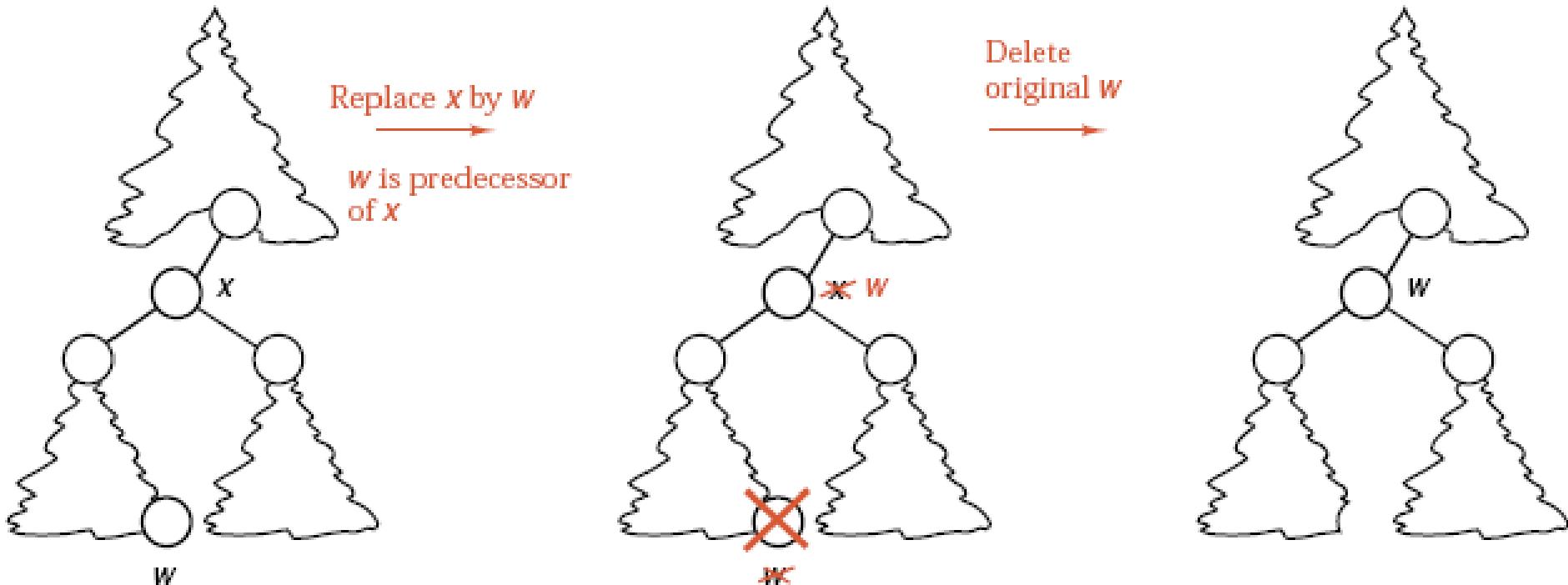
Trường hợp 3: X có đủ 2 con

- 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 :
Phản tử lớn nhất trên cây con trái
(tức là phản tử cực phải của cây con trái của x)

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

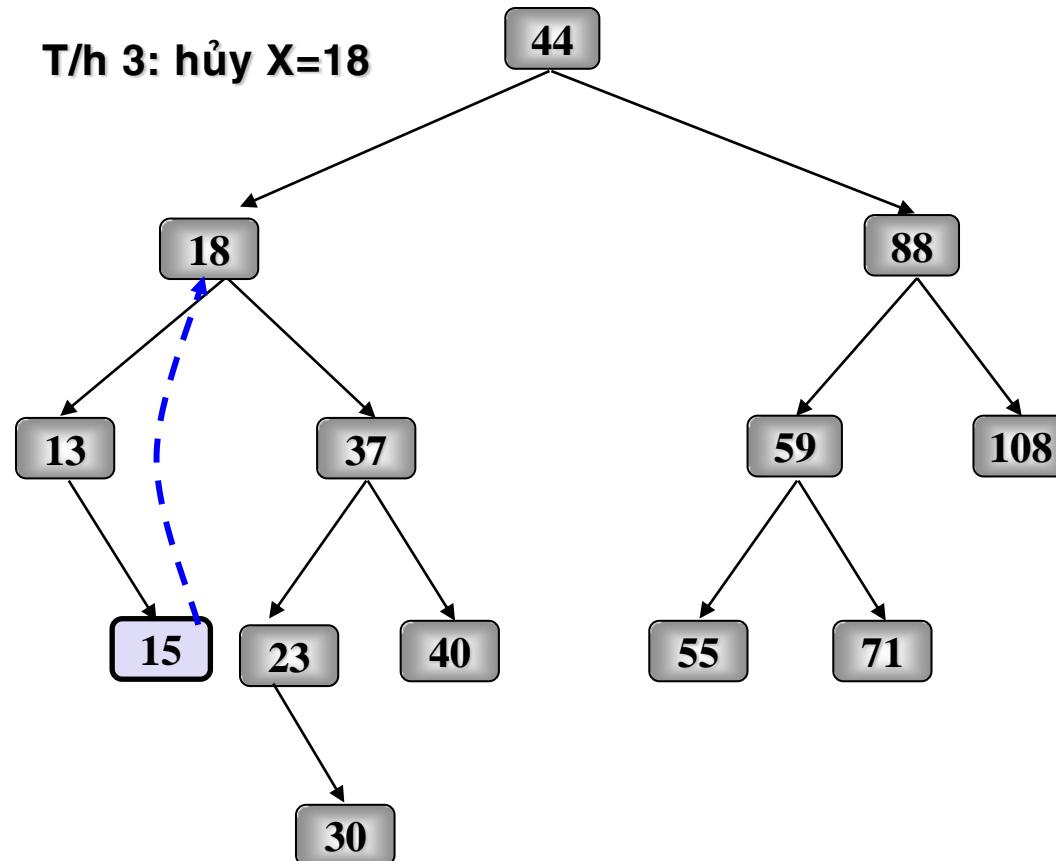


- Tìm w là **nút trước nút x trên phép duyệt cây inorder (LNR)** - chính là nút cực phải của cây con bên trái của x .
- Thay x bằng w
- Xóa nút w cũ (giống trường hợp 1 hoặc 2 đã xét)



- 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 :

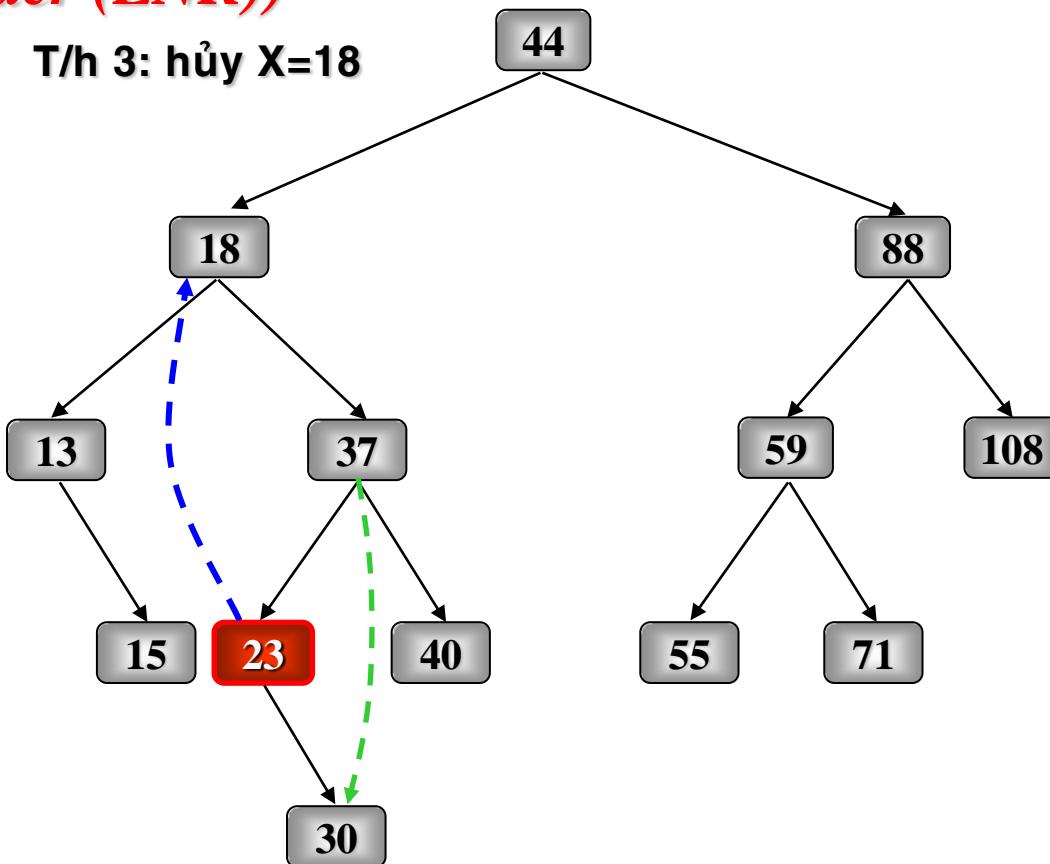
Nút thê mạng là *Phân tử nhỏ nhất trên cây con phải* (tức là phân tử cực trái của cây con phải của x)



- 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 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 trứ 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 $\log_2(n)$.