



28TECH  
Become A Better Developer

# GIỚI THIỆU CÂY NHỊ PHÂN



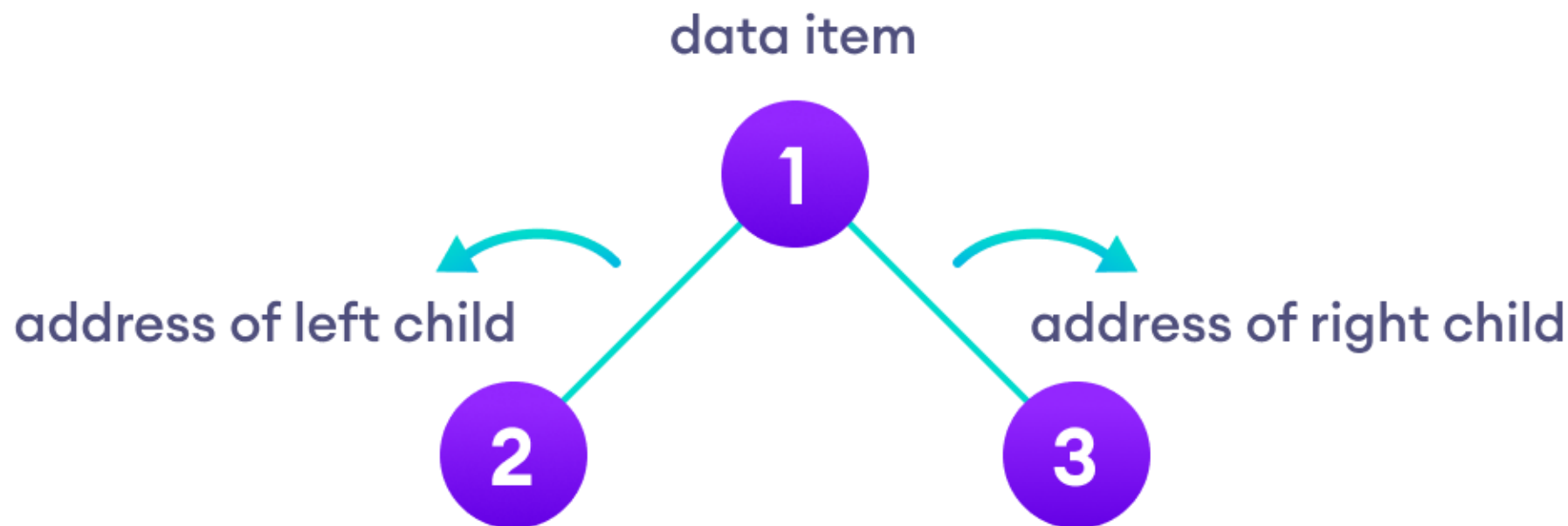
# 1. Giới thiệu cây nhị phân:



**Cây nhị phân** là một cấu trúc dữ liệu mà mỗi node cha có tối đa 2 node con thường gọi là node con bên trái và node con bên phải.



**Mỗi node trên cây nhị phân gồm 3 phần:** Phần dữ liệu, địa chỉ của node con bên trái, địa chỉ của node con bên phải.



## 2. Các khái niệm cơ bản:

- **Node gốc (root):** Node đầu tiên tạo nên cây nhị phân.
- **Node cha (parent):** Node A là node cha của node B nếu B là node con bên trái hoặc bên phải của node A.
- **Node lá (leaf):** Node không có node con.
- **Node trung gian (Internal):** Node có node con bên trái hoặc node con bên phải hoặc cả 2.
- **Node trước (Ancestor):** Node A gọi là node trước của node B nếu B thuộc cây con bên trái hoặc cây con bên phải với gốc A.
- **Node sau trái (left descendent):** Node B gọi là node sau trái của node A nếu B thuộc cây con bên trái với gốc A.
- **Node sau phải (right descendent):** Node B gọi là node sau phải củ node A nếu B thuộc cây con bên phải với gốc A.

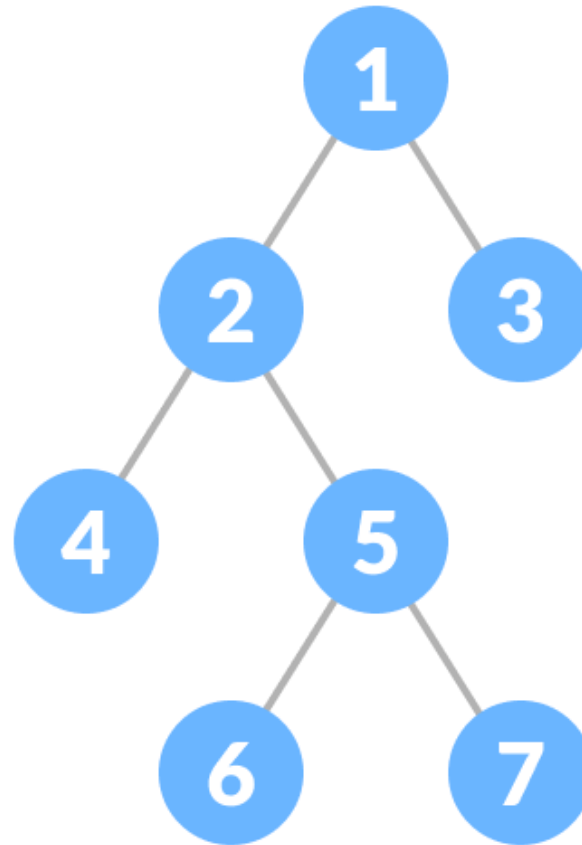
## 2. Các khái niệm cơ bản:

- **Bậc của node (degree):** Số cây con tối đa của một node, đối với cây nhị phân thì bậc của một node tối đa là 2
- **Mức của node (level):** Node gốc có mức 0, các node khác trên cây có mức bằng mức của node cha cộng thêm 1
- **Chiều sâu của node:** Bằng số cạnh trên đường đi từ node gốc tới node đang xét
- **Chiều sâu của cây:** Bằng mức cao nhất của 1 node lá trên cây hay chiều sâu lớn nhất của 1 node
- **Chiều cao của node:** Bằng số cạnh lớn nhất từ 1 node lá tới node đang xét
- **Chiều cao của cây:** Bằng chiều cao của node gốc

### 3. Các kiểu cây nhị phân:



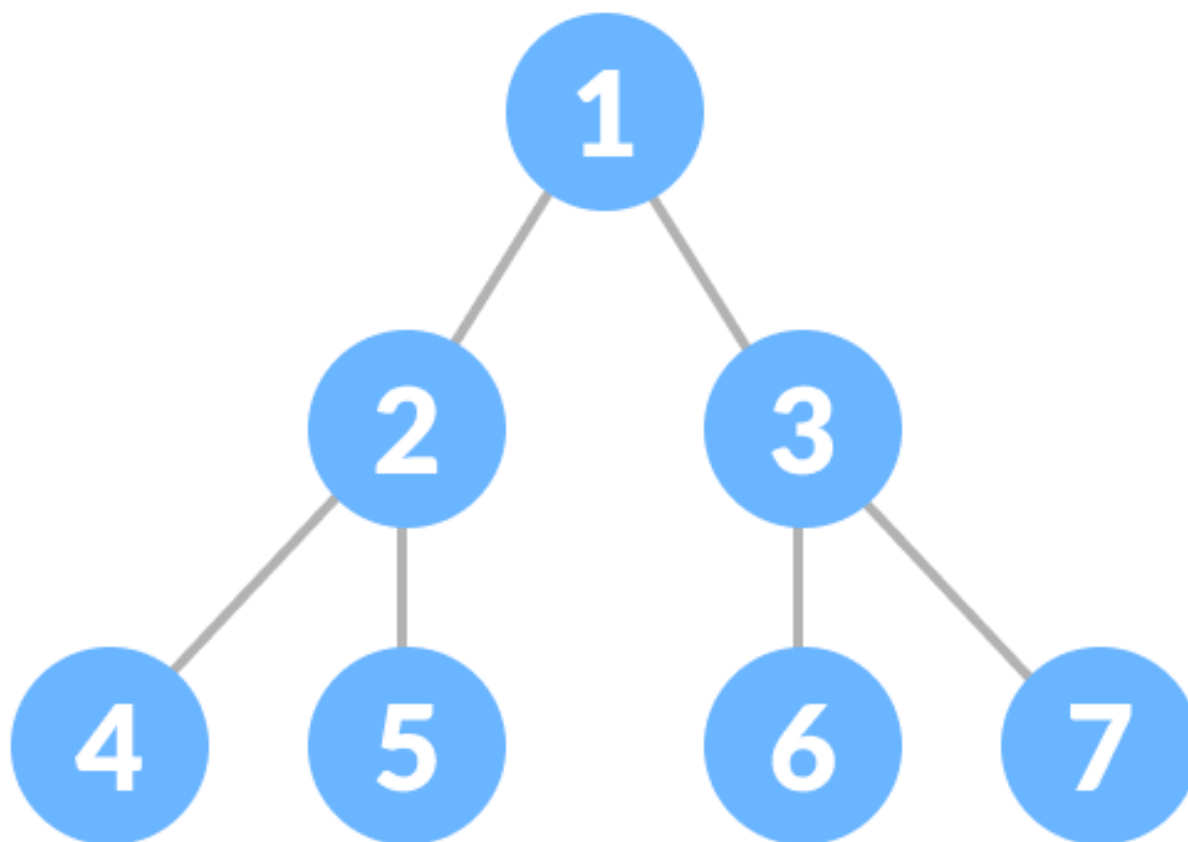
**Cây nhị phân đầy đủ (full binary tree):** Mỗi node cha có 0 hoặc 2 node con



### 3. Các kiểu cây nhị phân:



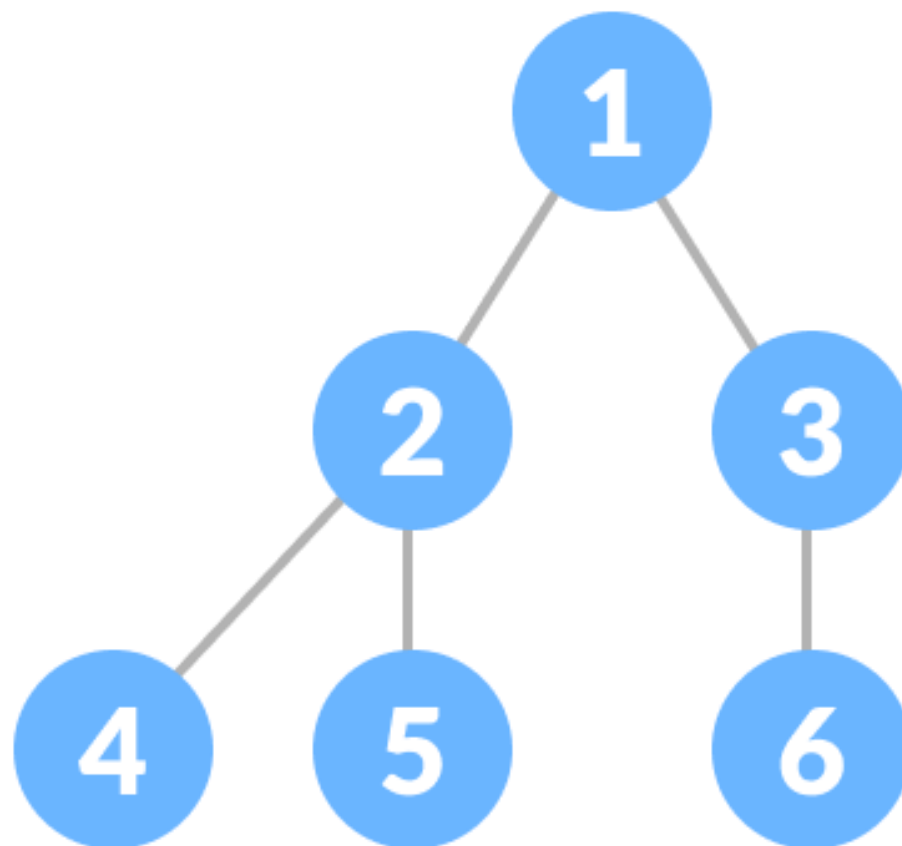
**Cây nhị phân hoàn hảo (Perfect binary tree):** Mỗi node trung gian đều có 2 con và mọi node lá đều có cùng mức.



### 3. Các kiểu cây nhị phân:



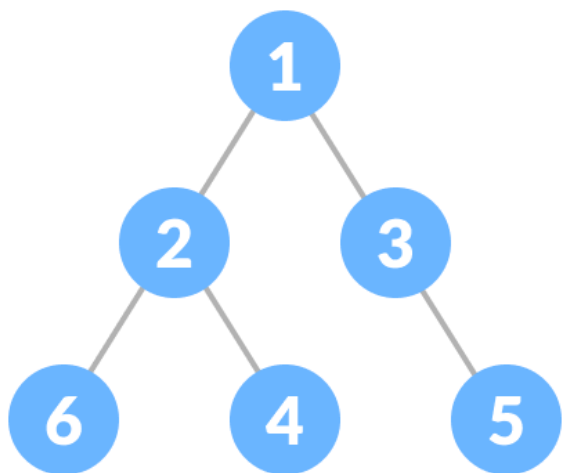
**Cây nhị phân hoàn chỉnh (complete binary tree):** Tất cả các node lá phải tinh gọn về bên trái; Node lá cuối cùng không nhất thiết phải có node anh em bên phải.



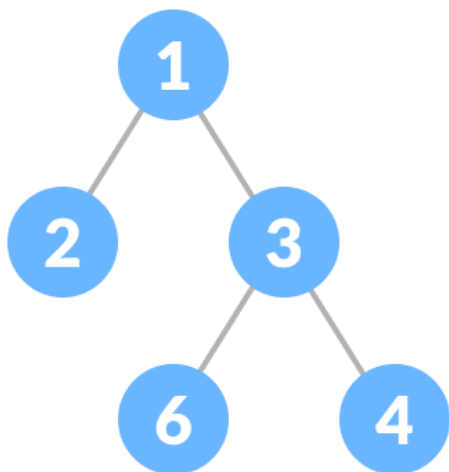
### 3. Các kiểu cây nhị phân:



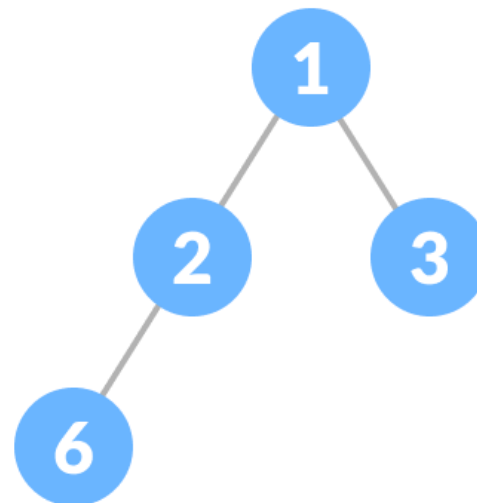
Phân biệt Full và Complete binary tree:



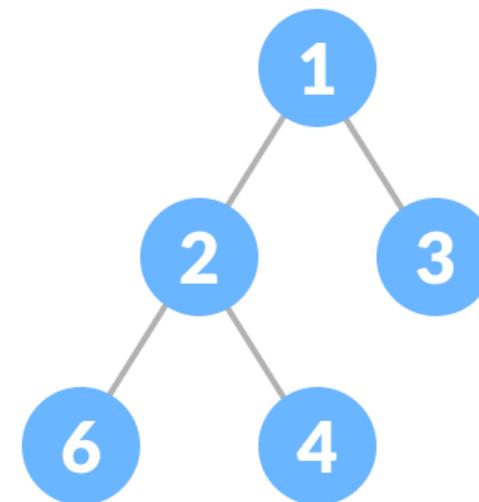
✗ Full Binary Tree  
✗ Complete Binary Tree



✓ Full Binary Tree  
✗ Complete Binary Tree



✗ Full Binary Tree  
✓ Complete Binary Tree



✓ Full Binary Tree  
✓ Complete Binary Tree



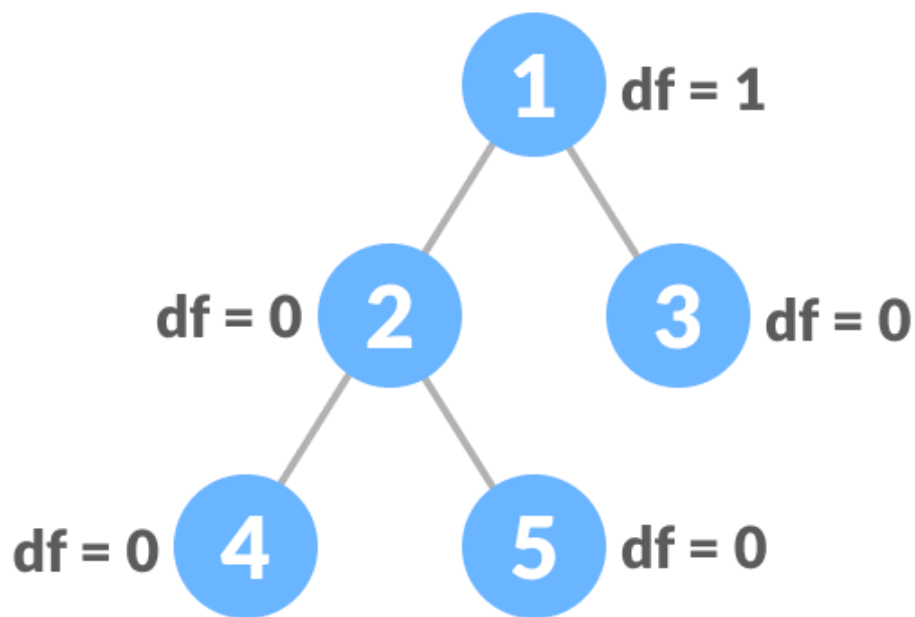


### 3. Các kiểu cây nhị phân:

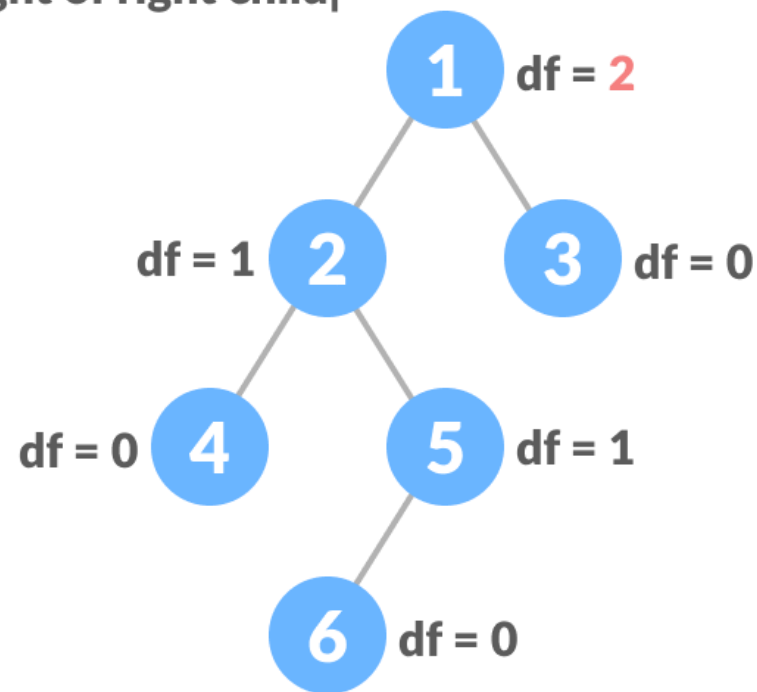


**Cây nhị phân cân bằng (Balanced binary tree):** Mọi node trên cây có độ cao của cây con bên trái và độ cao của cây con bên phải chênh lệch nhau không quá 1.

$$df = |\text{height of left child} - \text{height of right child}|$$



Cây nhị phân cân bằng



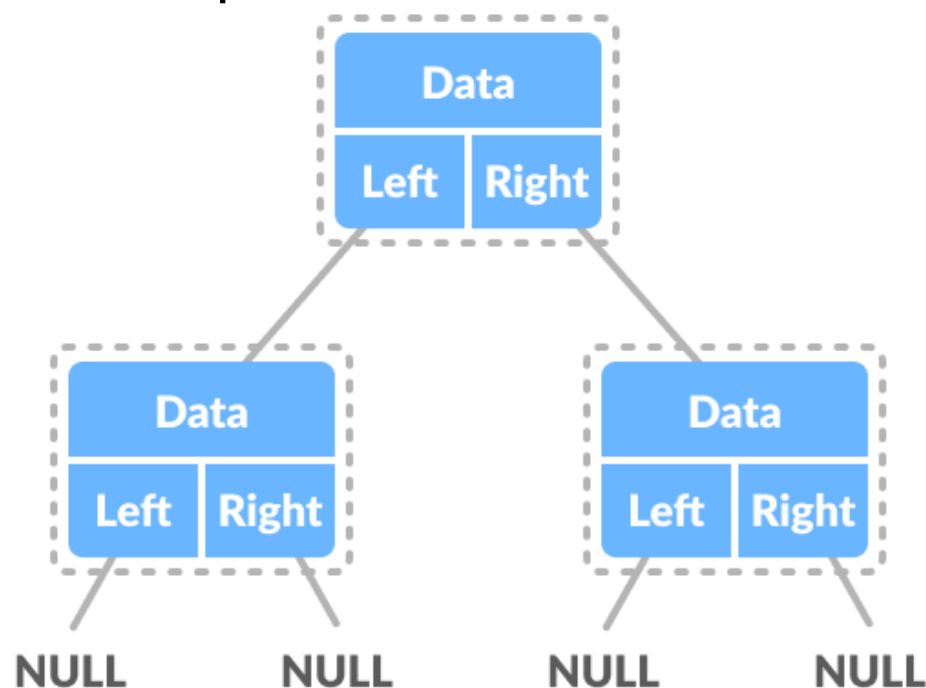
Cây nhị phân không cân bằng



## 4. Biểu diễn cây nhị phân:



**Để quản lý cây nhị phân** ta chỉ cần lưu địa chỉ của node gốc, tương tự như DSLK thì các node trên cây nhị phân đều được cấp phát động. Ngoài ra ta có thể biểu diễn cây nhị phân dưới dạng liên tục bằng cách sử dụng mảng 1 chiều, trong đó node gốc ở chỉ số 0, node cha ở chỉ số  $p$  thì node con bên trái ở chỉ số  $2p + 1$ , node con bên phải ở chỉ số  $2p + 2$ .



## 4. Biểu diễn cây nhị phân:

### Định nghĩa cấu trúc node

```
struct node{  
    int data;  
    node *left;  
    node *right;  
};
```

### Cấp phát node mới

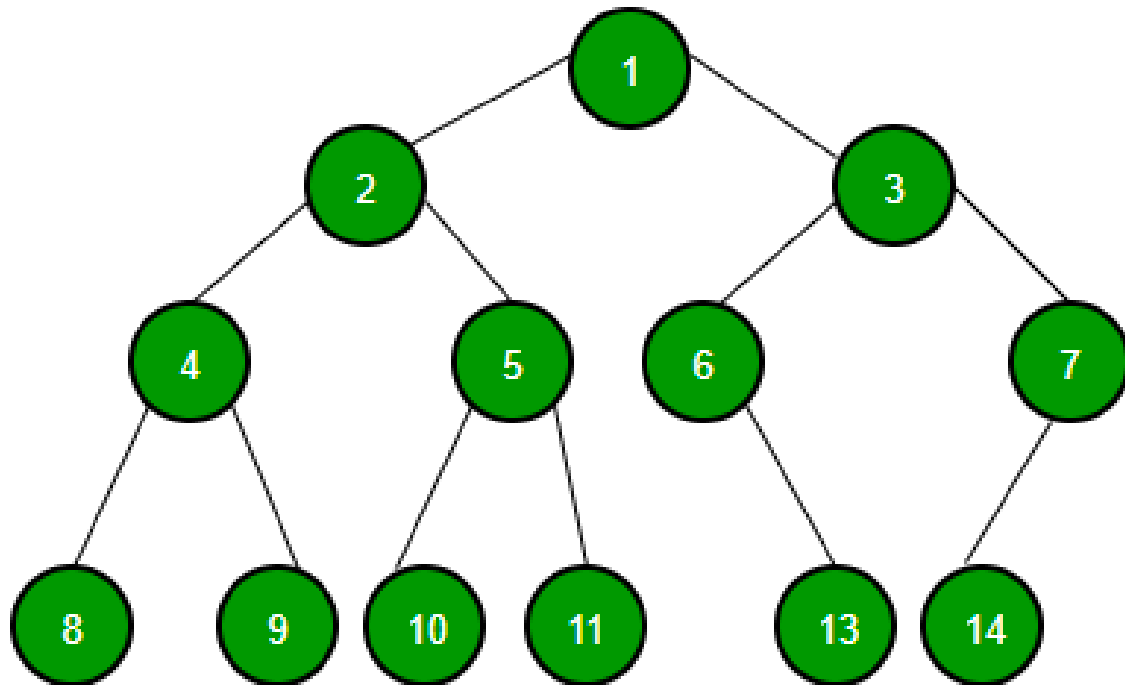
```
node *makeNode(int x){  
    node *newNode = new node;  
    newNode->data = x;  
    newNode->left = NULL;  
    newNode->right = NULL;  
    return newNode;  
}
```

## 5. Các thứ tự duyệt cây phổ biến:

### a. Duyệt giữa - Inorder (left - root - right):

Thứ tự duyệt:

8 - 4 - 9 - 2 - 10 - 5 - 11 - 1 - 6 - 13 - 3 - 14 - 7



Code:

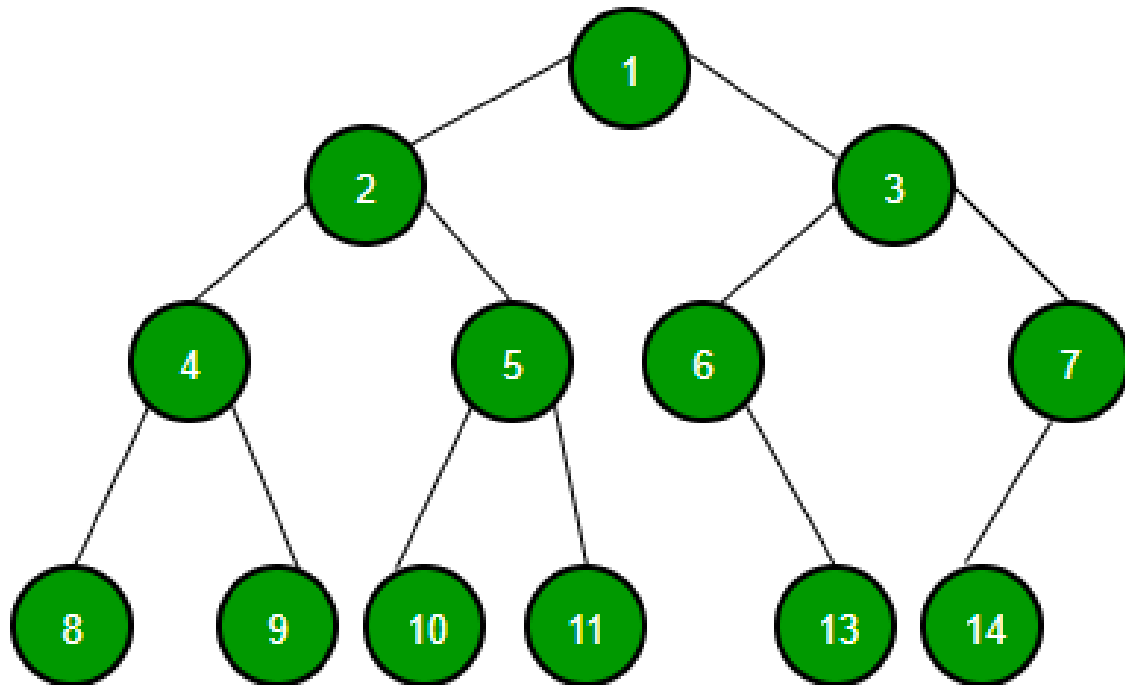
```
void inorder(node *root){  
    if(root != NULL){  
        inorder(root->left);  
        cout << root->data << ' ';  
        inorder(root->right);  
    }  
}
```

## 5. Các thứ tự duyệt cây phổ biến:

### b. Duyệt trước - Preorder (root - left - right):

Thứ tự duyệt:

1 - 2 - 4 - 8 - 9 - 5 - 10 - 11 - 3 - 6 - 13 - 7 - 14



Code:

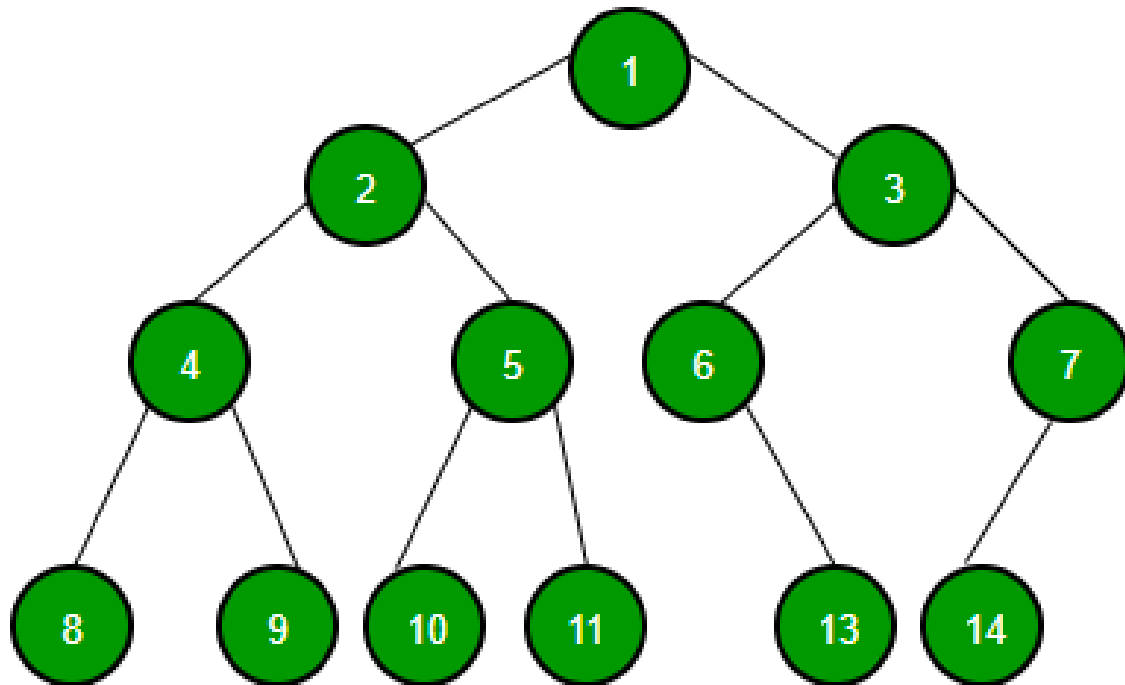
```
void preorder(node *root){  
    if(root != NULL){  
        cout << root->data << ' ';  
        preorder(root->left);  
        preorder(root->right);  
    }  
}
```

## 5. Các thứ tự duyệt cây phổ biến:

### c. Duyệt sau - Postorder (left - right - root):

Thứ tự duyệt:

8 - 9 - 4 - 10 - 11 - 5 - 2 - 13 - 6 - 14 - 7 - 3 - 1



Code:

```
void postorder(node *root){  
    if(root != NULL){  
        postorder(root->left);  
        postorder(root->right);  
        cout << root->data << ' ' ;  
    }  
}
```

## 5. Các thứ tự duyệt cây phổ biến:

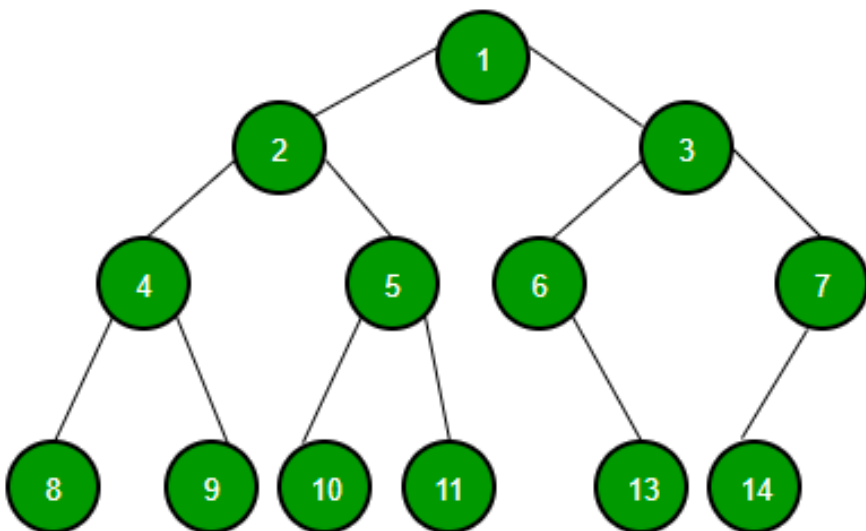
### d. Duyệt theo mức - Level Order:



**Duyệt theo mức** trên cây nhị phân chính là áp dụng BFS trên cây nhị phân.

**Thứ tự duyệt:**

**1 - 2 - 3 - 4 - 5 - 6 - 7 - 8 - 9 - 10 - 11 - 13 - 14**



**Code:**

```
void levelOrder(node *root){
    queue<node*> q;
    q.push(root);
    cout << root->data << ' ';
    while(!q.empty()){
        node *tmp = q.front(); q.pop();
        if(tmp->left != NULL){
            q.push(tmp->left);
            cout << tmp->left->data << ' ';
        }
        if(tmp->right != NULL){
            q.push(tmp->right);
            cout << tmp->right->data << ' ';
        }
    }
}
```



## 5. Các thứ tự duyệt cây phổ biến:

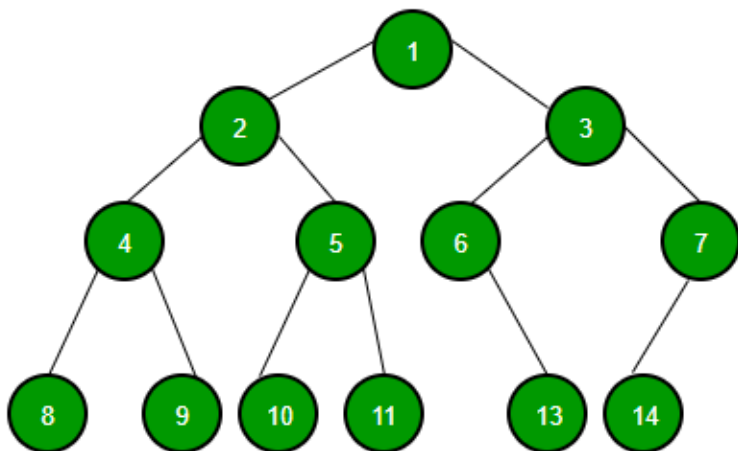
### e. Duyệt xoắn ốc - Spiral Order:



Để duyệt xoắn ốc ta sử dụng 2 ngăn xếp, ngăn xếp s1 để in các node theo thứ tự từ trái sang phải, ngăn xếp s2 để in các node theo thứ tự từ phải sang trái.

Thứ tự duyệt:

1 - 2 - 3 - 7 - 6 - 5 - 4 - 8 - 9 - 10 - 11 - 13 - 14



Code:

```
void spiral(node *root){
    stack<node*> s1, s2;
    s1.push(root);
    while(!s1.empty() || !s2.empty()){
        while(!s1.empty()){
            root *top = s1.top(); s1.pop();
            cout << top->data << ' ';
            if(top->right != NULL){
                s2.push(top->right);
            }
            if(top->left != NULL){
                s2.push(top->left);
            }
        }
        while(!s2.empty()){
            root *top = s2.top(); s2.pop();
            cout << top->data << ' ';
            if(top->left != NULL){
                s1.push(top->left);
            }
            if(top->right != NULL){
                s1.push(top->right);
            }
        }
    }
}
```





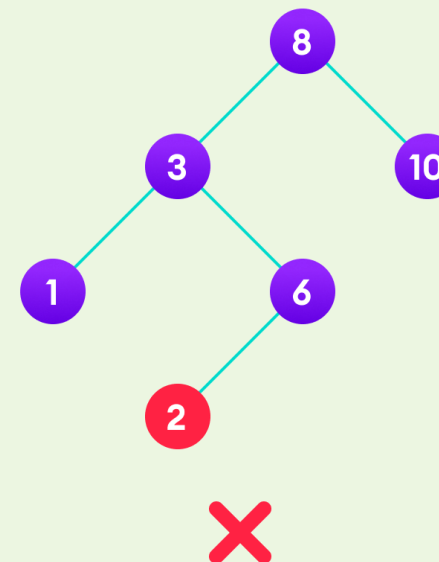
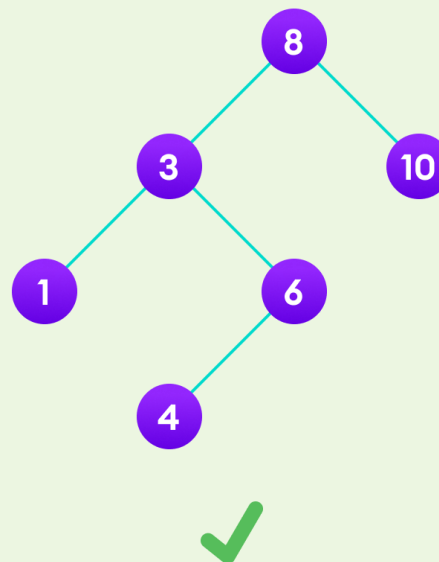
## 6. Cây nhị phân tìm kiếm:



**Cây nhị phân tìm kiếm - Binary search tree (BST)** là cây nhị phân có thể tìm kiếm sự xuất hiện của 1 phần tử trên cây với độ phức tạp  $\log N$ .

### Các tính chất của BST:

- Tất cả node con thuộc cây con bên trái nhỏ hơn node gốc.
- Tất cả node con thuộc cây con bên phải lớn hơn node gốc.
- Cây con bên trái và bên phải cũng là cây BST.



## 6. Cây nhị phân tìm kiếm:

### a. Thao tác tìm kiếm:



Do tính chất của BST nên khi tìm kiếm ta có thể nhanh chóng xác định được giá trị cần tìm kiếm thuộc cây con bên trái hay cây con bên phải của node gốc.

#### Có 3 trường hợp:

**Trường hợp 1:** Giá trị cần tìm kiếm bằng giá trị của node đang xét, ta kết luận tìm thấy.

**Trường hợp 2:** Nếu giá trị cần tìm kiếm nhỏ hơn giá trị node đang xét, tìm kiếm ở cây con bên trái.

**Trường hợp 3:** Tìm kiếm ở cây con bên phải nếu giá trị tìm kiếm lớn hơn giá trị node gốc.

#### Code:

```
bool search(node *root, int key){  
    if(root == NULL) return false;  
    if(root->data == key) return true;  
    else if(root->data < key)  
        return search(root->right, key);  
    else  
        return search(root->left, key);  
}
```



## 6. Cây nhị phân tìm kiếm:

### b. Thao tác chèn:



Khi chèn một giá trị mới vào cây, cây vẫn phải duy trì được tính chất là cây nhị phân tìm kiếm.

#### Thuật toán:

- Kiểm tra giá trị cần chèn với giá trị của node gốc, nếu giá trị cần chèn nhỏ hơn giá trị node gốc thì đi sang cây con bên trái, ngược lại đi sang cây con bên phải.
- Khi gặp node lá, chèn giá trị cần chèn vào node con bên trái hoặc bên phải tùy theo giá trị.

#### Code:

```
node *insert(node *root, int key){  
    if(root == NULL){  
        return makeNode(key);  
    }  
    if(key < root->data){  
        root->left = insert(root->left, key);  
    }  
    else if(key > root->data){  
        root->right = insert(root->right, key);  
    }  
    return root;  
}
```

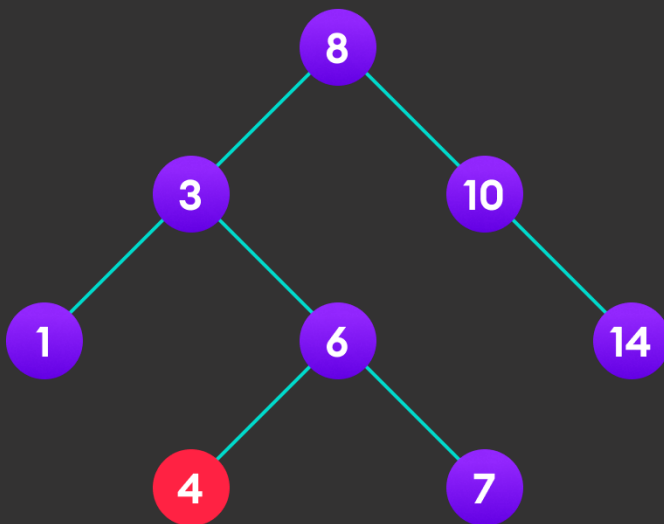


## 6. Cây nhị phân tìm kiếm:

### c. Thao tác xóa:

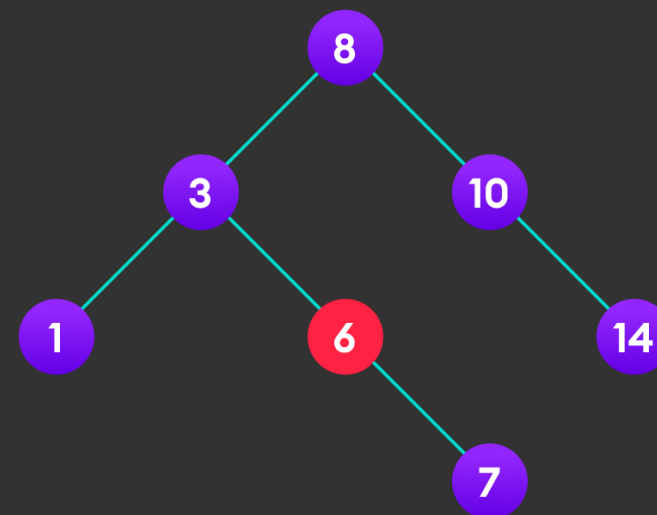
#### Trường hợp 1:

Node cần xóa là node lá, ta chỉ cần giải phóng node này:



#### Trường hợp 2:

Node cần xóa có 1 node con, khi đó gán giá trị của node cần xóa bằng giá trị của node con sau đó giải phóng node con:



## 6. Cây nhị phân tìm kiếm:

### c. Thao tác xóa:

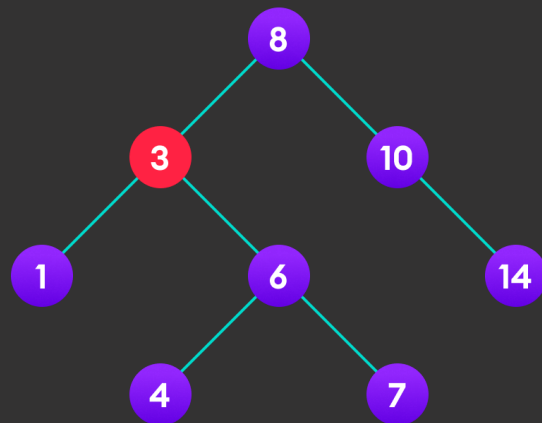
#### Trường hợp 3:

##### Node cần xóa có 2 node con

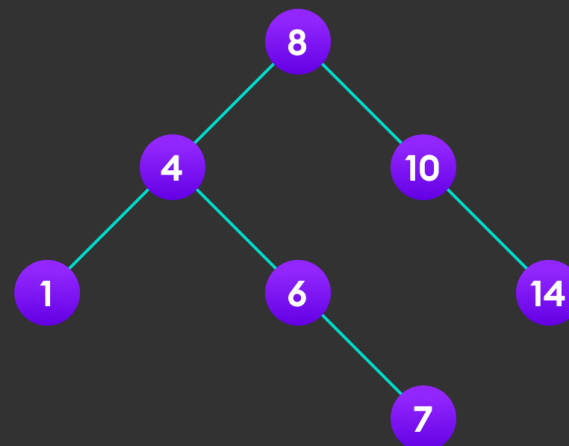
**Bước 1.** Tìm node con X nhỏ nhất lớn hơn node cần xóa

**Bước 2.** Gán giá trị node cần xóa thành giá trị của node X

**Bước 3.** Xóa node X khỏi cây



##### Sau khi xóa



## 6. Cây nhị phân tìm kiếm:

### c. Thao tác xóa:

#### Code:

```
node* minNode(node *root){
    node *temp = root;
    while(temp != NULL && temp->left != NULL){
        temp = temp->left;
    }
    return temp;
}

node* deleteNode(node *root, int key){
    if(root == NULL) return root;
    if(key < root->data){
        root->left = deleteNode(root->left, key);
    }
    else if(key > root->data){
        root->right = deleteNode(root->right, key);
    }
    else{
        if(root->left == NULL){
            node *tmp = root->right;
            delete root;
            return tmp;
        }
        else if(root->right == NULL){
            node *tmp = root->left;
            delete root;
            return tmp;
        }
        else{
            node *tmp = minNode(root);
            root->data = tmp->data;
            root->right = deleteNode(root->right, tmp->data);
        }
    }
}
```

