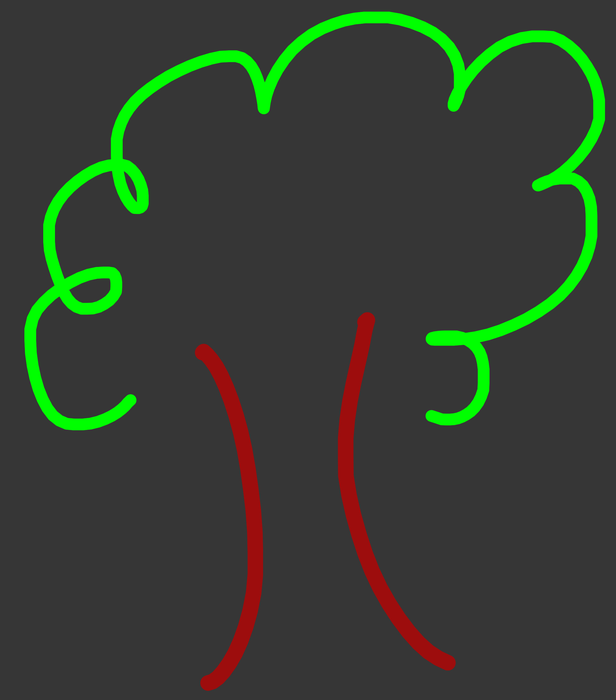
All the Best!!!

Binary Tree

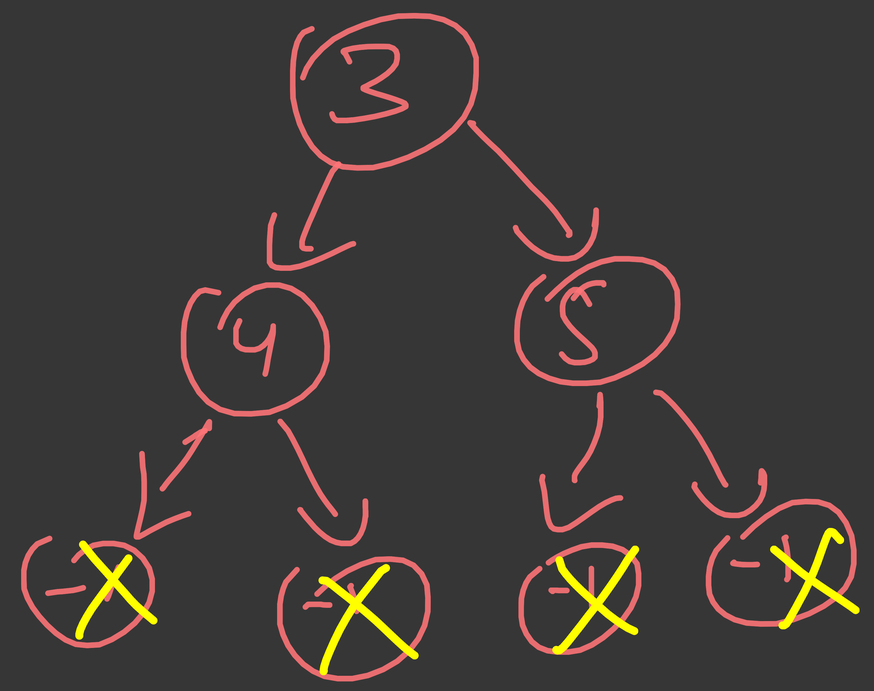


Binary Tree Node Class

```
class Node{
    public:
        int data;
        Node*left;
        Node*right;

    Node(int d){
        data = d;
        left = right = NULL;
    }
};
```

Build Tree :-

3 4 -1 -1 5 -1 -1

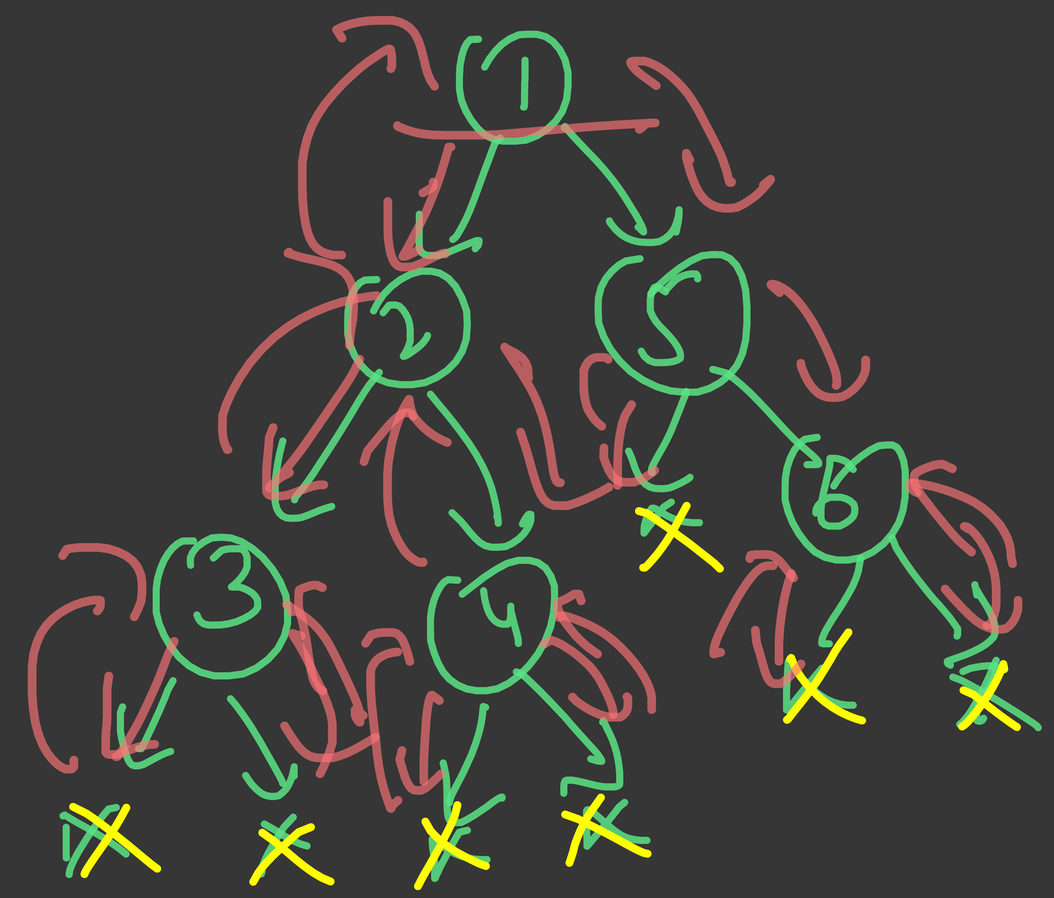



```
void buildTree(){
    // input data
    int d;
    cin >> d;

    // if input is -1 then simply return NULL
    if(d==-1)
        return NULL;

    // create a new Tree Node with data d
    root = new Node(d);

    // recursion call to left and right subtree
    root->left = buildTree();
    root->right = buildTree();

    // at the end return root
    return root;
}
```

Print Tree :- Pre-Order Traversal

```
void PreOrder(Node*root){
    if(root==NULL)
        return;

    cout << root->data << " ";
    PreOrder(root->left);
    PreOrder(root->right);
}
```

1 2 3 4 5 6

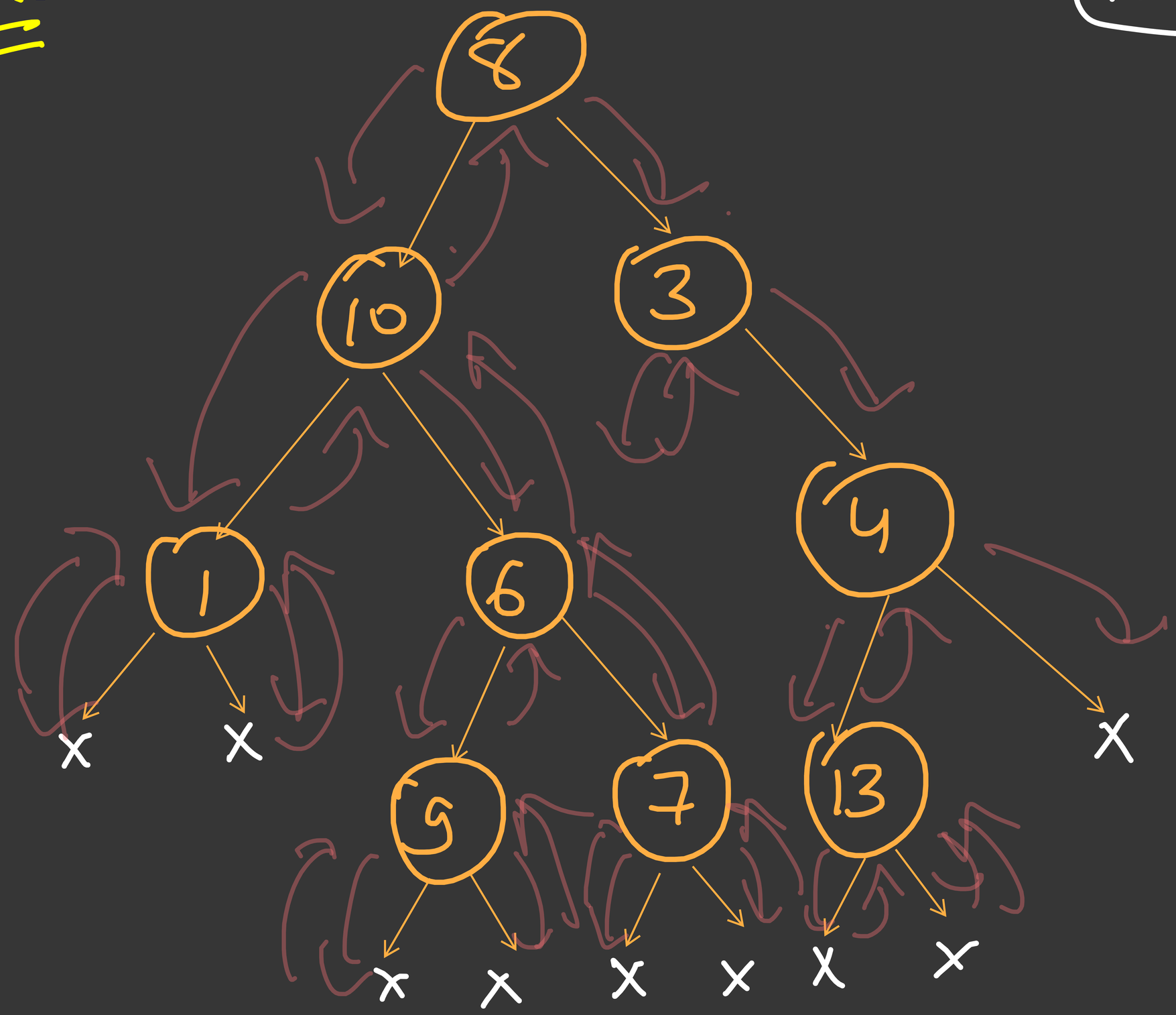


In-Order Traversal

```
void InOrder(Node*root){
    if(root==NULL)
        return;
    InOrder(root->left);
    cout << root->data << " ";
    InOrder(root->right);
}
```

Inorder

X → NULL

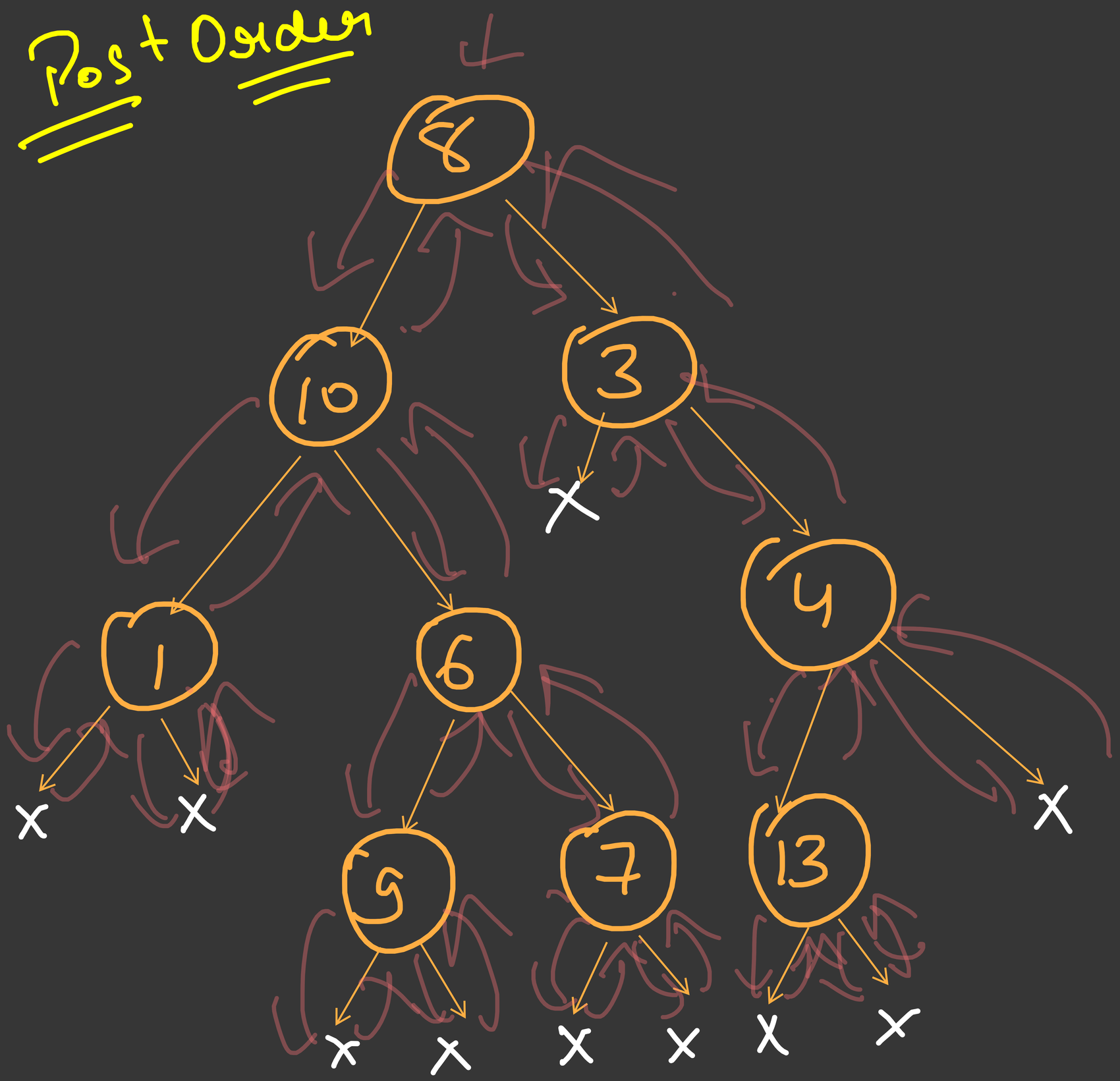


1
10
9
6
7
8
3
13
4

Post Order Traversal :-

```
void PostOrder(Node*root){
    if(root==NULL)
        return;
    PostOrder(root->left);
    PostOrder(root->right);
    cout << root->data << " ";
}
```

Post Order

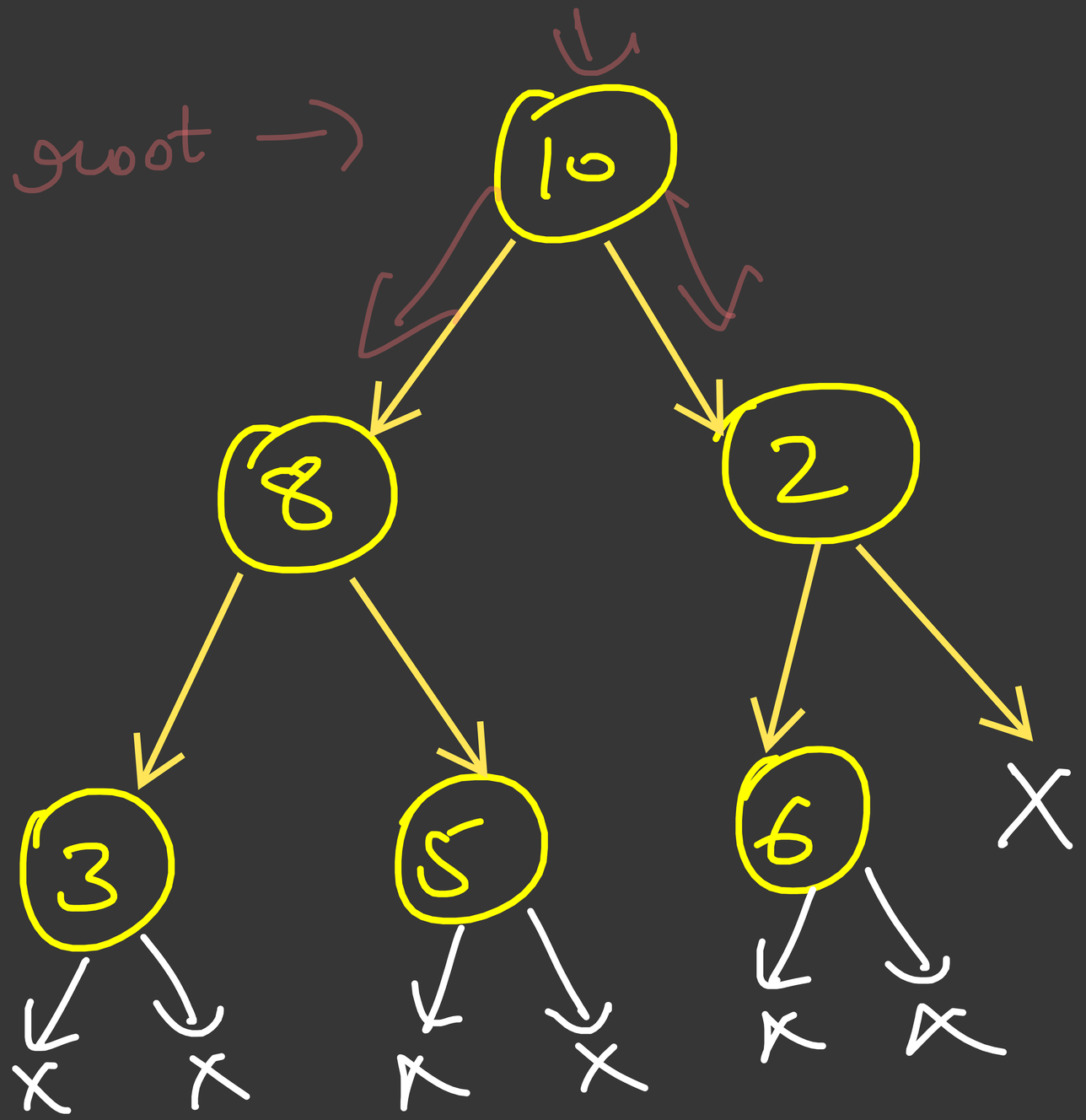


1
9
7
6
10
13
4
3
8

Pre-Order Iterative :-

```
iterativePreorder(node)


if (node = null)
  return
s -> empty stack
s.push(node)
while (not s.isEmpty())
  node -> s.pop()
  visit(node)
  if (node.right != null)
    s.push(node.right)
  if (node.left != null)
    s.push(node.left)
```

```cpp
void PreOrderIterative(Node*root){
    if(root==NULL)
        return;

    // create a stack
    stack<Node*> stack;
    stack.push(root);

    while(!stack.empty()){
        Node*curr = stack.top();
        stack.pop();
        cout << curr->data << " ";

        // if right subtree is present
        // then fill into the stack
        if(curr->right)
            stack.push(curr->right);
        // if left subtree is present
        // then fill into the stack
        if(curr->left)
            stack.push(curr->left);
    }
}
```



Stack

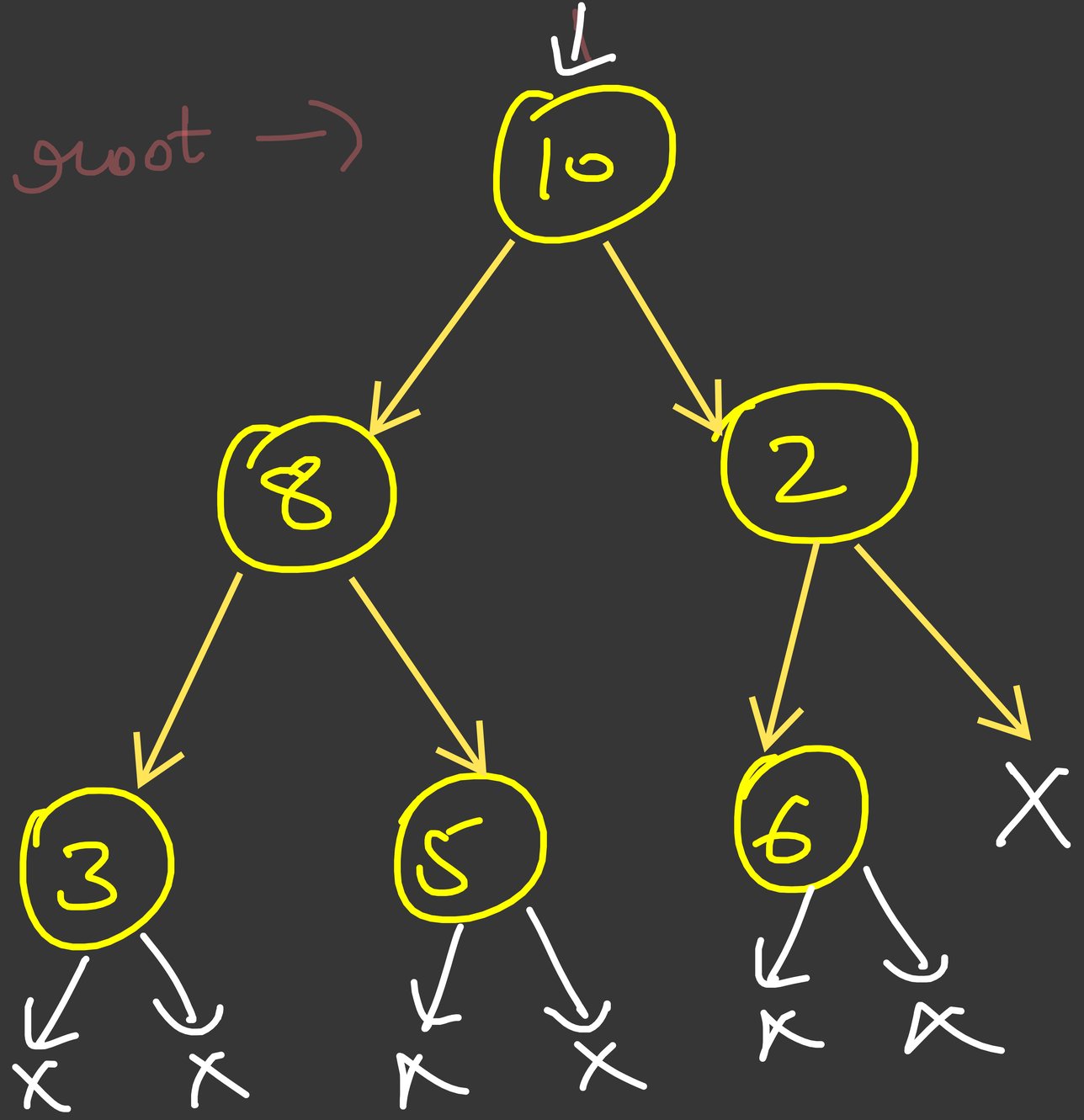
10 |
~~10~~ | 2 | 8 |
2 | ~~8~~ | 5 | 3 |
2 | ~~5~~ | ~~3~~ |
~~2~~ | 6 |
~~6~~ |

O/P:- 10 8 3 5 2 6

In Order Traversal Iterative :-

Algo :-

```
iterativeInorder(node)


  s -> empty stack
  while (not s.isEmpty() or node != null)
    if (node != null)
      s.push(node)
      node -> node.left
    else
      node -> s.pop()
      visit(node)
      node -> node.right
```



O/P:- 3, 8, 5, 10, 6, 2

Stack

10 | 8 | ~~3~~
10 | ~~8~~ | 5
10 | ~~5~~ |
~~10~~ | 2 | 6
2 | ~~6~~
~~2~~

```cpp
void InOrderIterative(Node*root){
    // create a stack
    stack<Node*> stack;

    // assign root to curr node
    Node*curr = root;

    while(curr!=NULL || !stack.empty()){
        if(curr!=NULL){
            stack.push(curr);
            curr = curr->left;
        }
        else{
            curr = stack.top();
            cout << curr->data << " ";
            stack.pop();
            curr = curr->right;
        }
    }
}
```

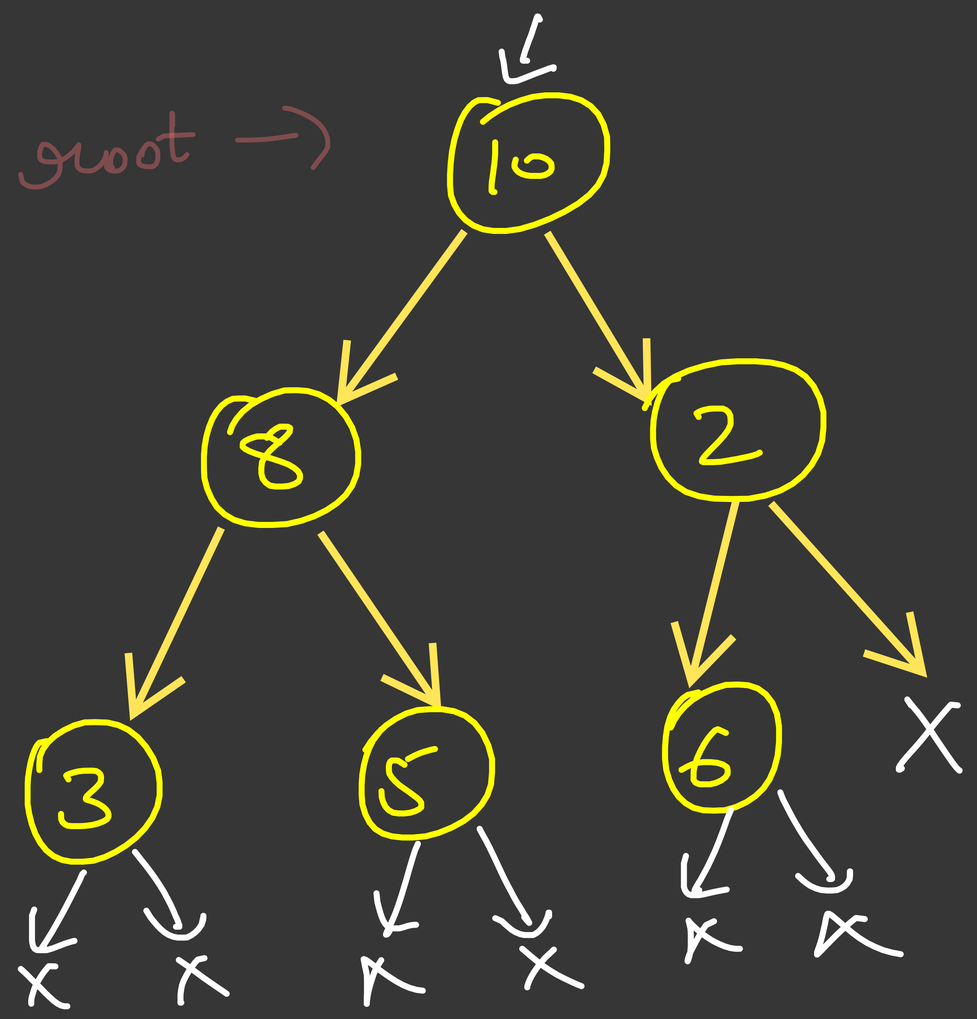
PostOrder Iterative Traversal

Algorithm

```
iterativePostorder(node)


s -> empty stack
t -> output stack
while (not s.isEmpty())
  node -> s.pop()
  t.push(node)

  if (node.left <> null)
    s.push(node.left)

  if (node.right <> null)
    s.push(node.right)

while (not t.isEmpty())
  node -> t.pop()
  visit(node)
```

for storing

for O/P



O/P:- 3, 8, 5, 10, 6, 2

Stack	ans
~~10~~ | 8 | 2	10 |
8 | ~~2~~ | 6	10 | 2
8 | ~~6~~	10 | 2 | 6
~~8~~ | 3 | 5	10 | 2 | 6 | 8
3 | ~~5~~	10 | 2 | 6 | 8 | 5
~~3~~	10 | 2 | 6 | 8 | 5 | 3

3 5 8 6 2 10

reverse

```
void PostOrderIterative(Node*root) {
    if (root == NULL)
        return;

    stack<Node*>s;
    s.push(root);
    // answer stack will be the int type
    stack<int> ans;

    while (!s.empty()) {

        Node*curr = s.top();
        s.pop();
        ans.push(curr->data);

        // if any node present in left, then push into the stack
        if (curr->left)
            s.push(curr->left);
        // if any node presnet in right, then push into the stack
        if (curr->right)
            s.push(curr->right);
    }
    // now print the answer stack
    while (ans.empty() == false) {
        cout << ans.top() << " ";
        ans.pop();
    }
}
```

Max Depth or Height of Binary Tree

Algorithm:-

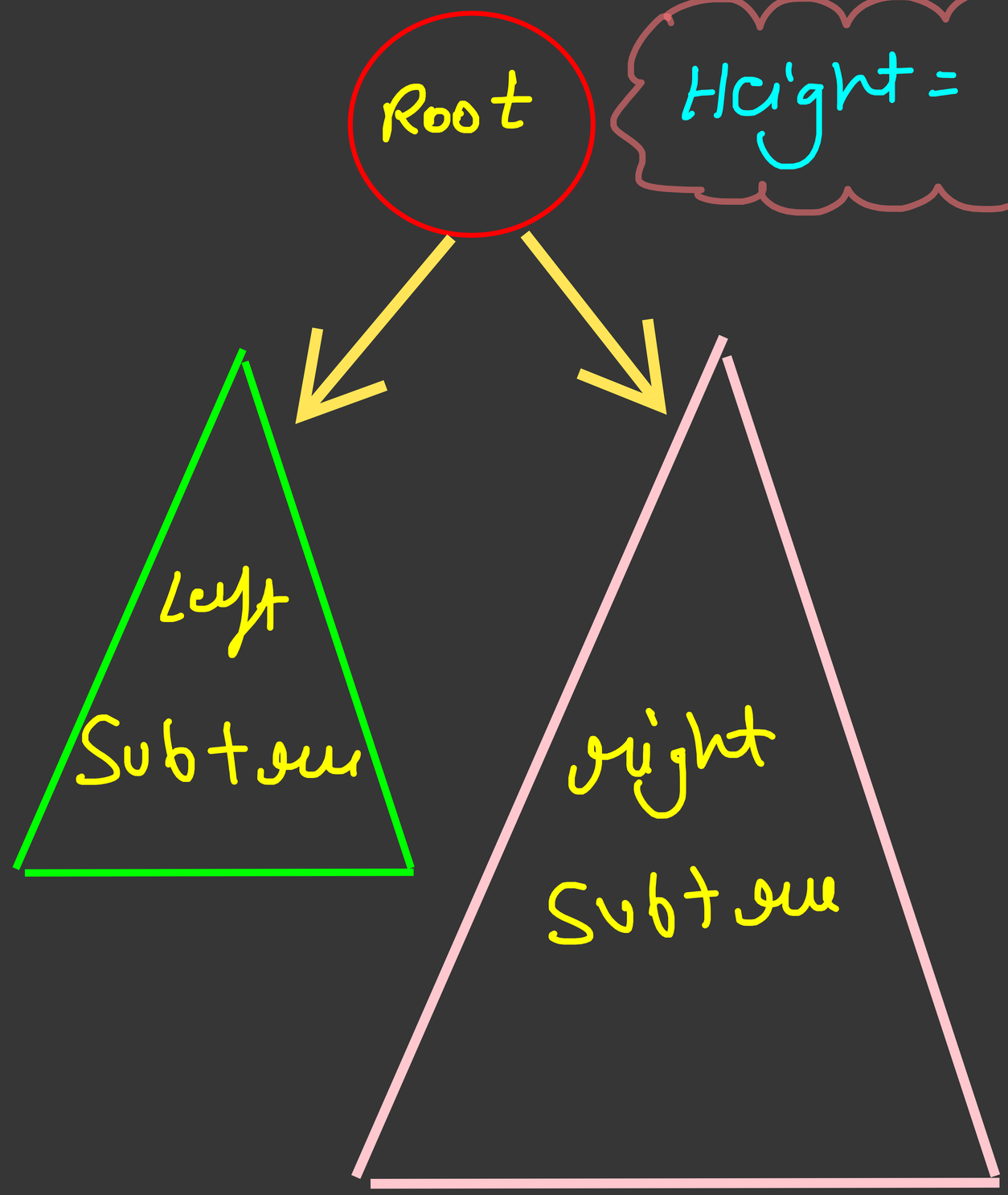
```
1. If tree is empty then return -1
2. Else
    (a) Get the max depth of left subtree recursively  i.e.,
        call maxDepth( tree->left-subtree)
    (a) Get the max depth of right subtree recursively  i.e.,
        call maxDepth( tree->right-subtree)
    (c) Get the max of max depths of left and right
        subtrees and add 1 to it for the current node.
        max_depth = max(max dept of left subtree,
                        max depth of right subtree)
                        + 1
    (d) Return max_depth
```

Source Code:-

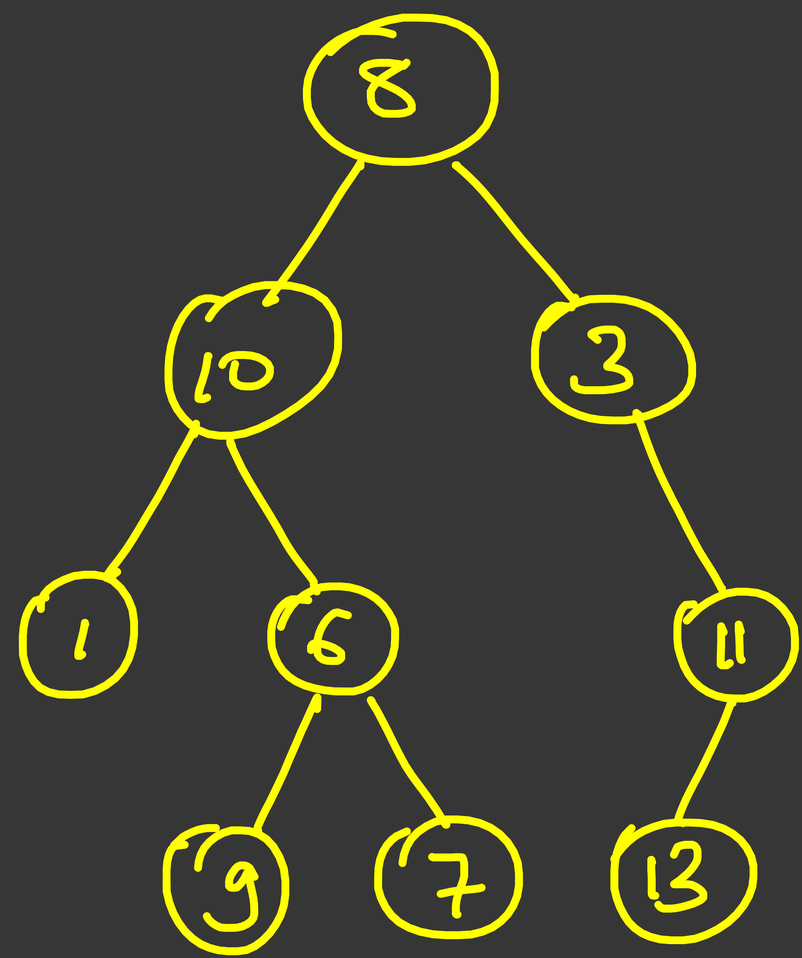
```
int Height(Node*root){
    // if there is no root, height will be zero
    if(root==NULL)
        return 0;
    // left subtree height
    int leftHeight = Height(root->left);
    // right subtree height
    int rightHeight = Height(root->right);
    // max of leftsubtree and rightsubtree will be our answer
    re
```

Concept behind Height of Tree:-

Height = 1 + max(left, right)



Print k^{th} level of Binary Tree :-



* I/P :-

  $k = 3$

* O/P :-

  1 6 11

```
void printKthLevel(Node*root){
    if(root==NULL)
        return;
    // if k == 1; that mean this is the desired level
    // for which we have to print
    if(k==1) {
        cout << root->data << " ";
        return;
    }

    // call left and right subtree of root
    printKthLevel(root->left, k-1);
    printKthLevel(root->right, k-1);
    return;
}
```

Concept behind Printing kth Level :-

$\boxed{k=3}$

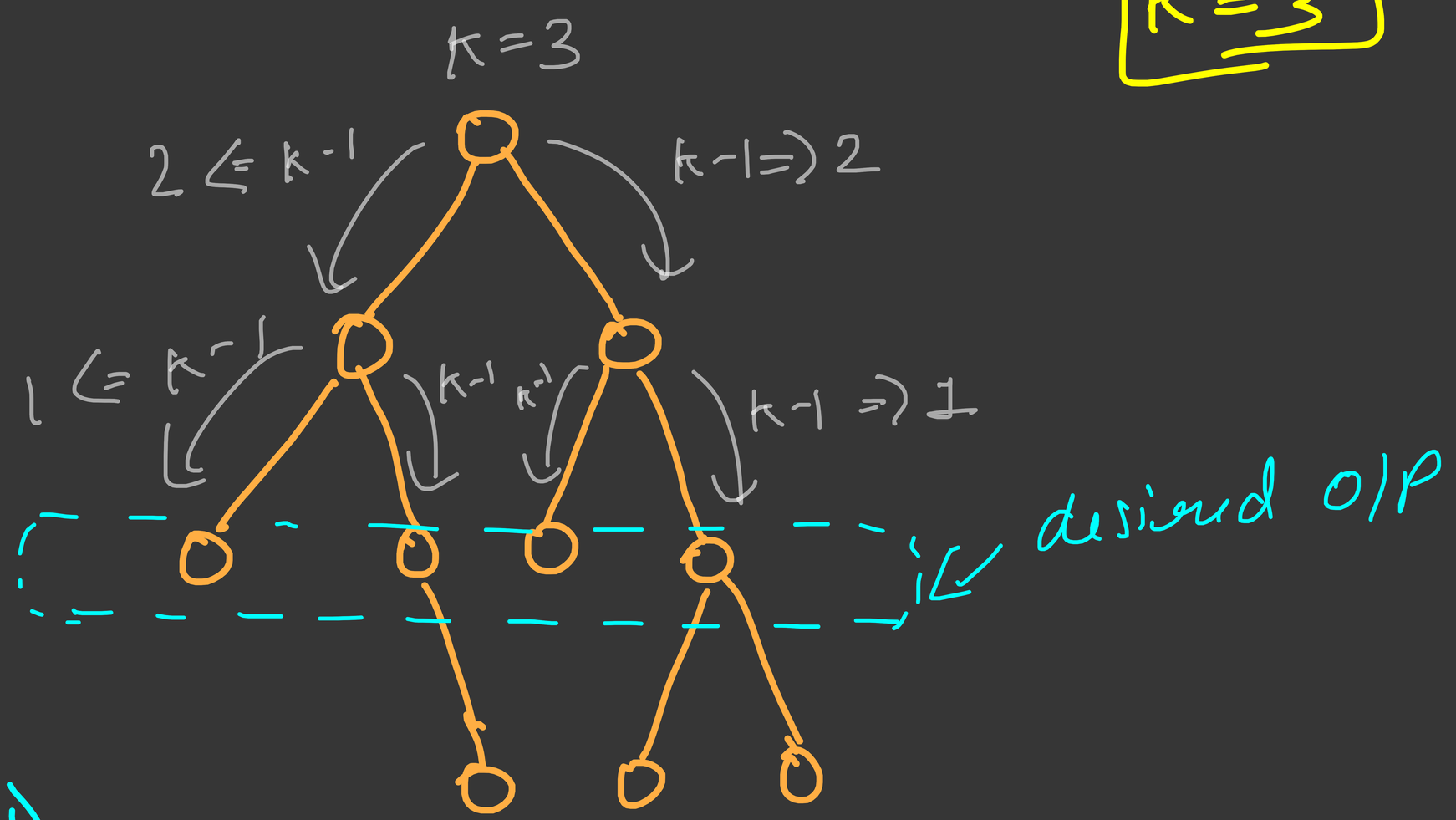
$k=3$

$\downarrow k-1$

$k=2$

$\downarrow k-1$

$k=1$




```
*
if (k==1)
    print(data)
```

Level Order Traversal :-

① find the Height of Tree

```
int H = Height(root);
```

② Iterate 0 to H-1

```
for (int i=0; i< H; i++)
    Print nth Level (root, i);
    Cout << endl;
}
```

③ return;

```
void LevelOrder(Node*root){
    if(root==NULL)
        return;

    // computer the height of tree
    int H = Height(root);

    // iterate 0 to H-1 for printing level by level
    for(int i = 0; i < H; i++){
        printKthLevel(root, i);
        cout << endl;
    }
    return;
}
```