DATA STRUCTURE AND PROGRAMMING II

Tree data structure

Advanced data structure

Outline

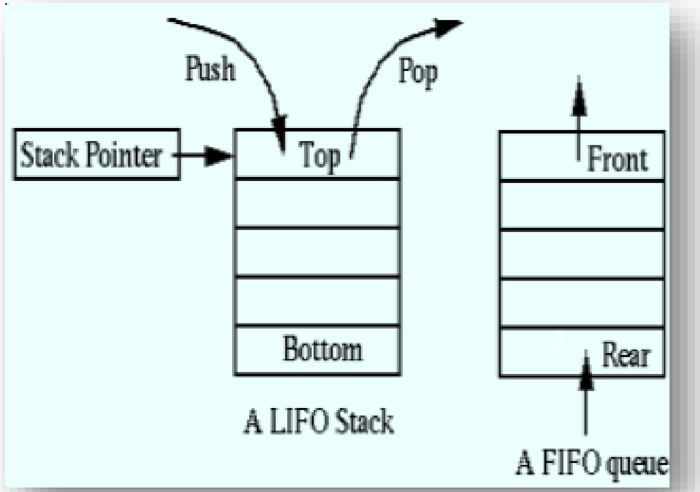
- Data structure
  - Linear Vs. Non linear
- What is Tree? Binary tree? Binary search tree (BST)?
- What are Tree operations?
- Traversal of Tree
- How to implement Tree in C++
- Examples

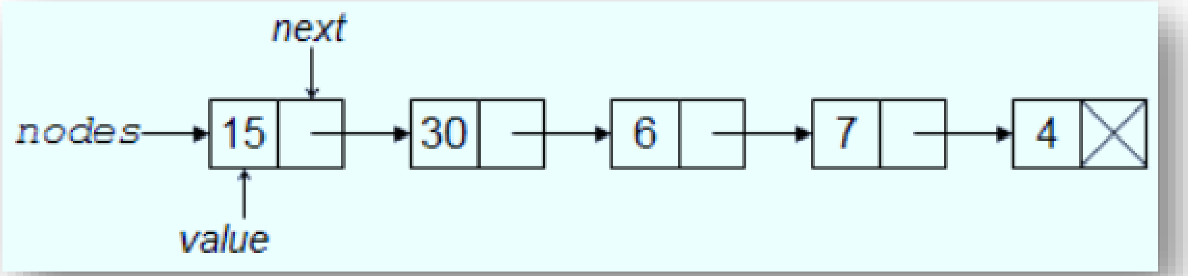
Data structure

❑ Linear Data Structure

- Data structure helps to **store and organize data** in computer
- **Linear data structure** stores data in such a way that the data can be accessed **sequentially (continuous)**
  - Array, linked list, stack queue

a[index]	a[0]	a[1]	a[2]	a[3]
Data element	5	7	2	-1
Address	0xefabc1	0xefabc3	0xefabc5	0xefabc7





Linear Vs. Non-linear data structure

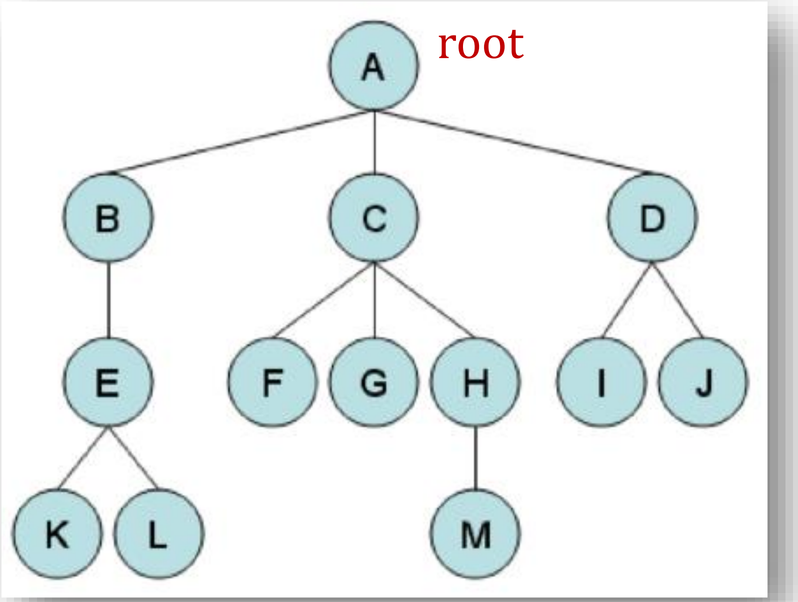
❑ Comparison

Factor	Linear data structure	Non-linear data structure
How data is stored	Data elements construct a sequence of a linear list.	Does not arrange data consecutively but arrange in sorted order.
Traversal of data	▪ Data elements are visited sequentially ▪ Traversal of element is easy	▪ Traversal of data elements and insertion/deletion are not done sequentially ▪ Traversal of element is difficult
Implementation	Simple	Complex
Levels	Single level of elements	Multiple levels of elements (hierarchical)
Memory utilization	Ineffective	Effective
Example	Array, linked list, stack, queue	Tree, graph

Tree

❑ Definition

- A tree is a hierarchical (non-linear) data structure defined on a set of elements called nodes
- A tree can be empty or composed of nodes
  - The top-level node is called ***root***, while other nodes are sub-tree



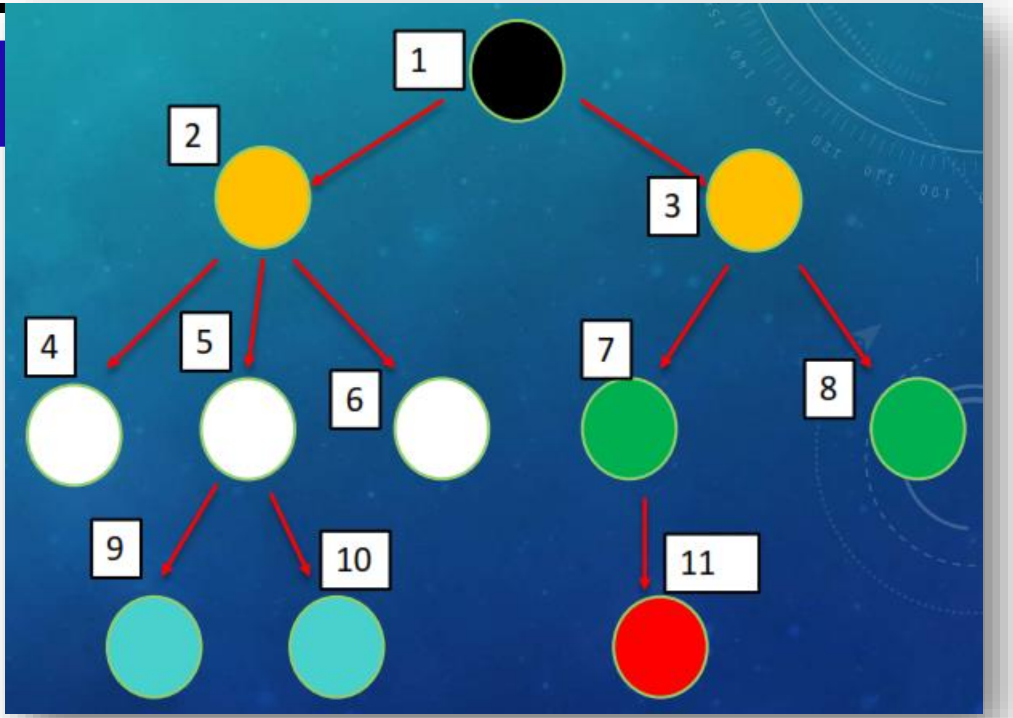
An example of a tree

Tree

❑ Relation of Tree

- **Root** : top element
- **Children** : have same parents,

  grant parents, great grant parents, ...
- **Parents** : have children
- **Siblings** : have same parent
- **Leaf** : is element that has no children
  - Remark: In particular, leaf element has pointer points to NULL



How many leaves are there?

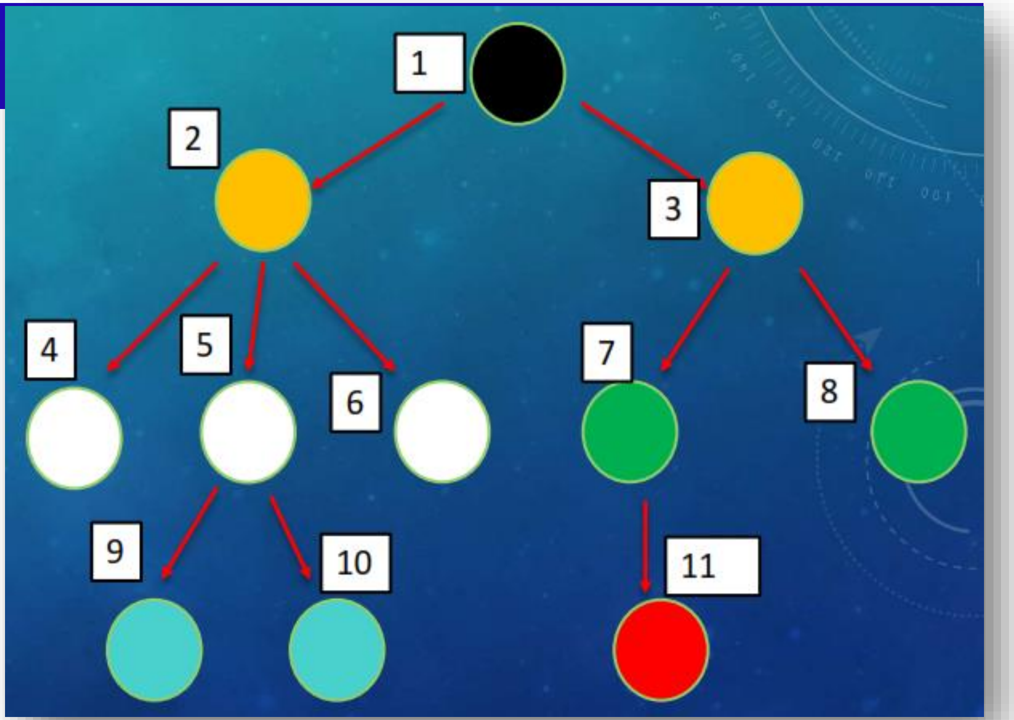
=> 6

What are they?

=> 4, 9, 10, 6, 11, 8

Relation of Tree

❑ Edge Vs. Depth Vs. Height

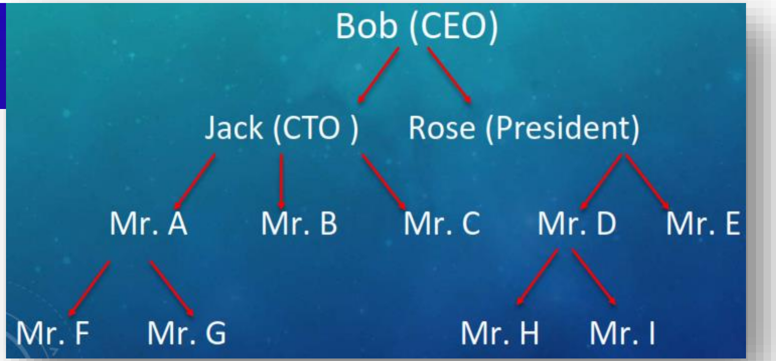


- **Edge (path)**
  - An edge is a line connected two nodes together
  - If a tree have N nodes, then it has (N-1) edges
- **Depth of node x**
  - Depth of node x is number of edges from x to root
  - Note: Depth of root is 0
- **Height of node x**
  - Height of node x is number of edges on longest path from x to a leaf
- Remark:
  - Height of a tree = depth of a tree = longest path of the tree
  - Size of a tree is the number of elements (nodes)
  - Branch is any path from the root to a leaf

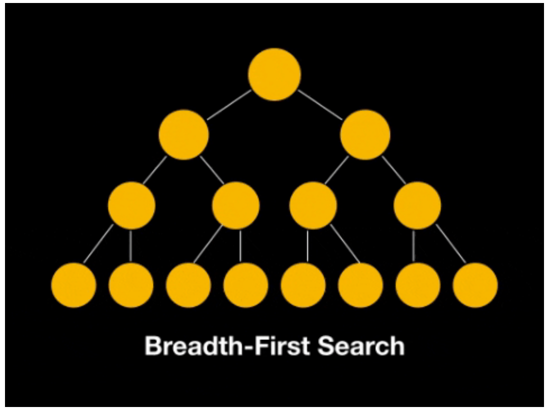
How many edges? => 10 edges
What is the depth of node 7? => 2
What is the height of node 1? => 3

Tree applications

❑ Some examples



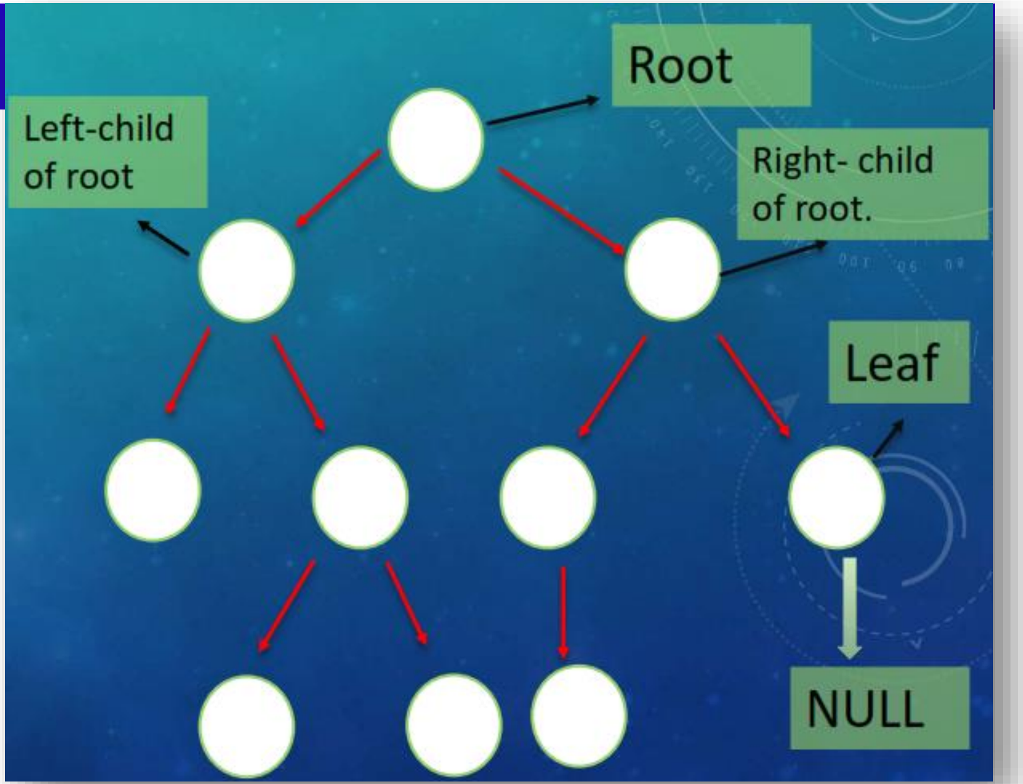
1. Store hierarchical data (file system)
2. Organize data for quick search, insertion, deletion
   - Binary search tree (BST)
3. Dictionary
4. Network routing algorithm



Binary Tree

❑ Definition

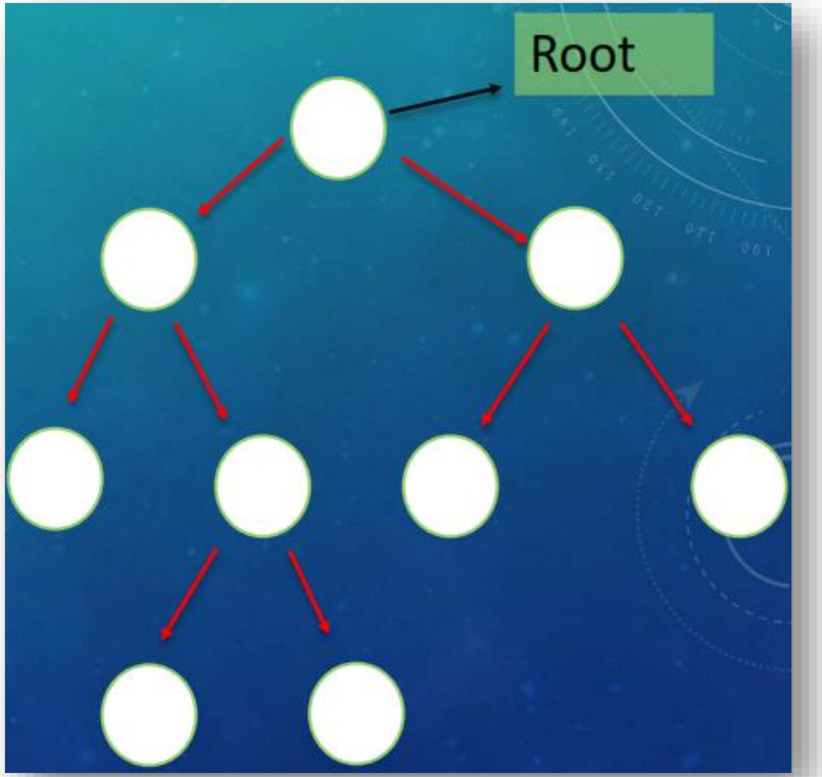
- Each node can have at most 2 children
- A node has left and/or right child
- A leaf node has no left or right child.
  - It has only NULL
- Types of Binary Tree
  1. Strict/proper/full binary tree
  2. Complete binary tree
  3. Perfect binary tree



Strict/proper/full Binary Tree

❑ Definition

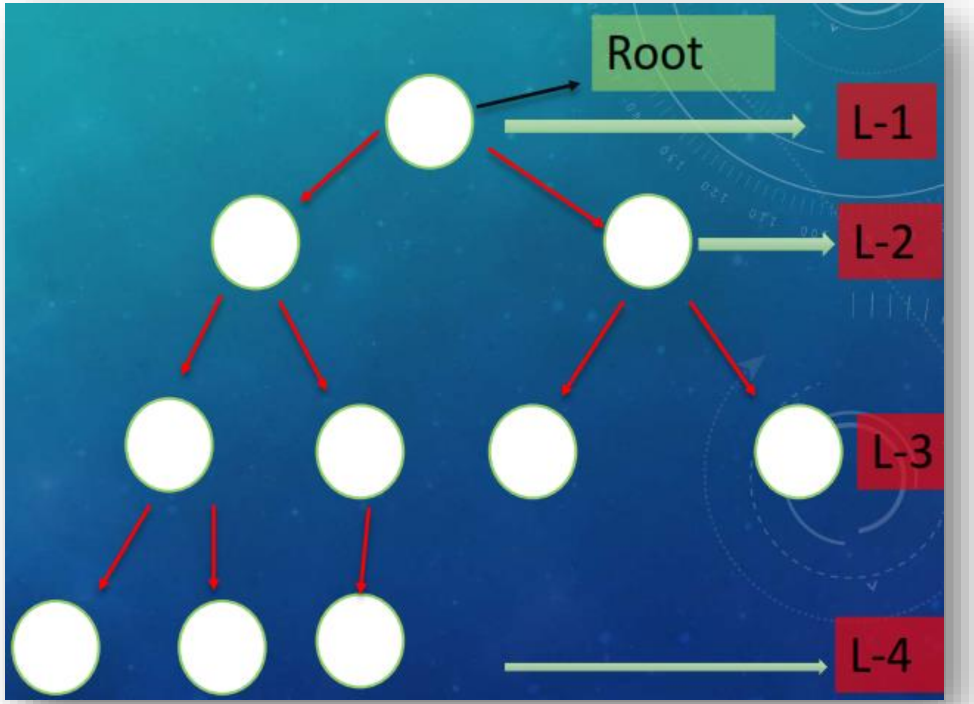
- Each node can have either 2 or 0 child
- It can not have only one left or right child

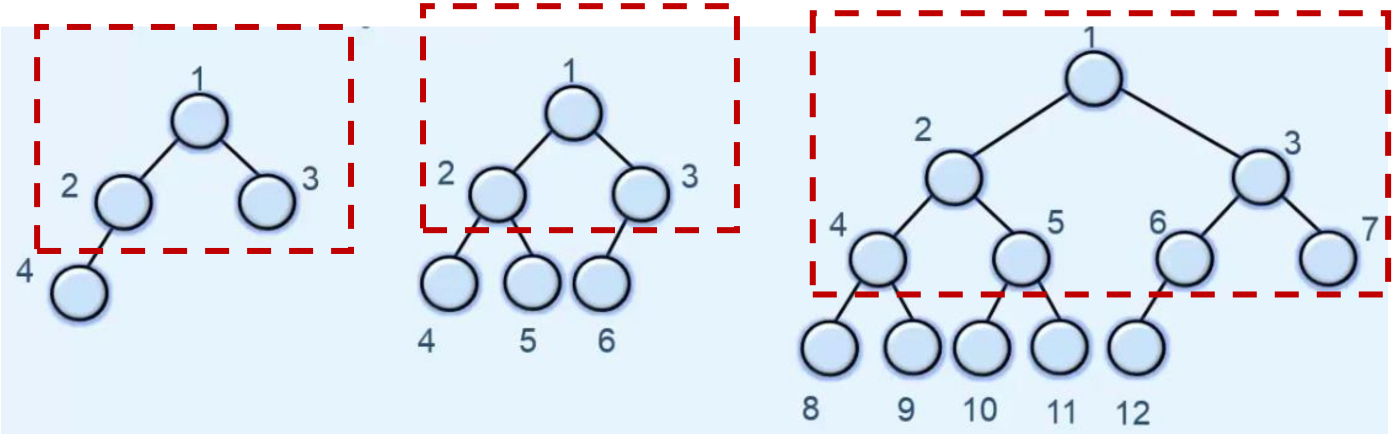


Complete Binary Tree

❑ Definition

- A complete binary tree is a binary tree in which every level, except possibly the last level, is completely filled and all nodes are as far left as possible.

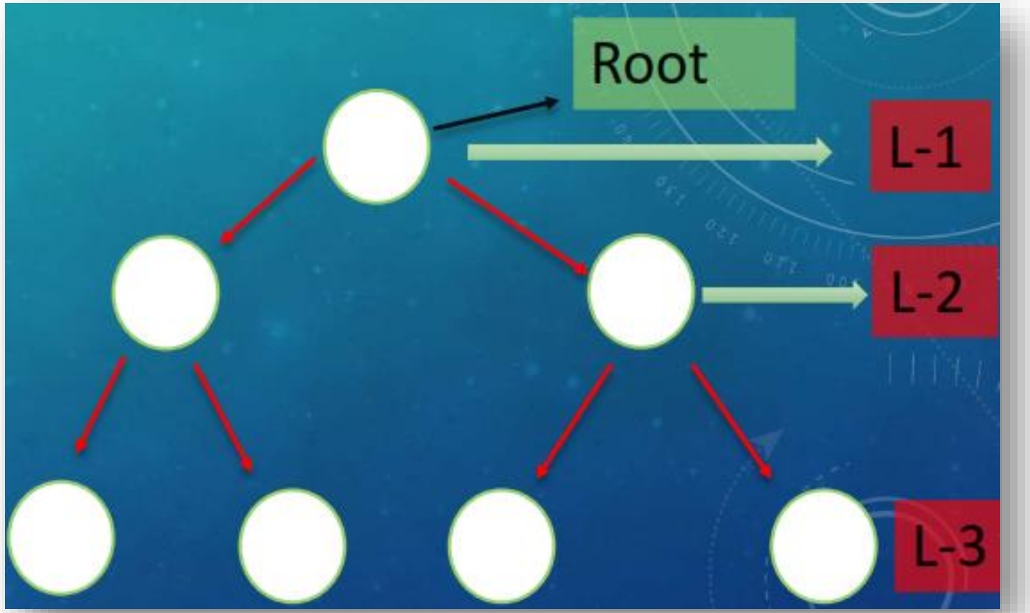




Perfect Binary Tree

❑ Definition

- All levels are completed filled and balanced



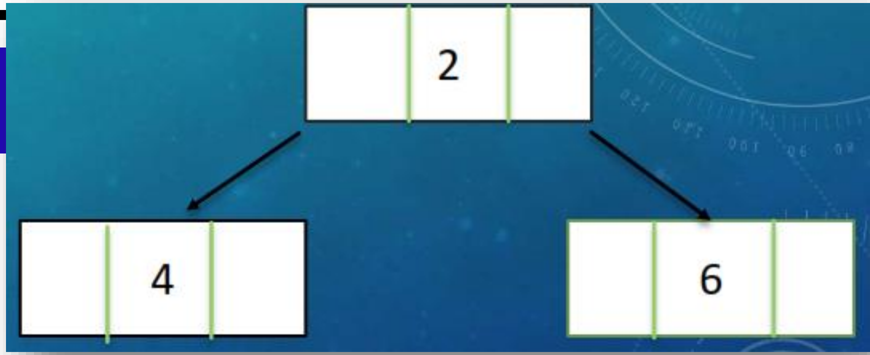
Tree Implementation

Tree implementation

❑ There are 2 types of implementation

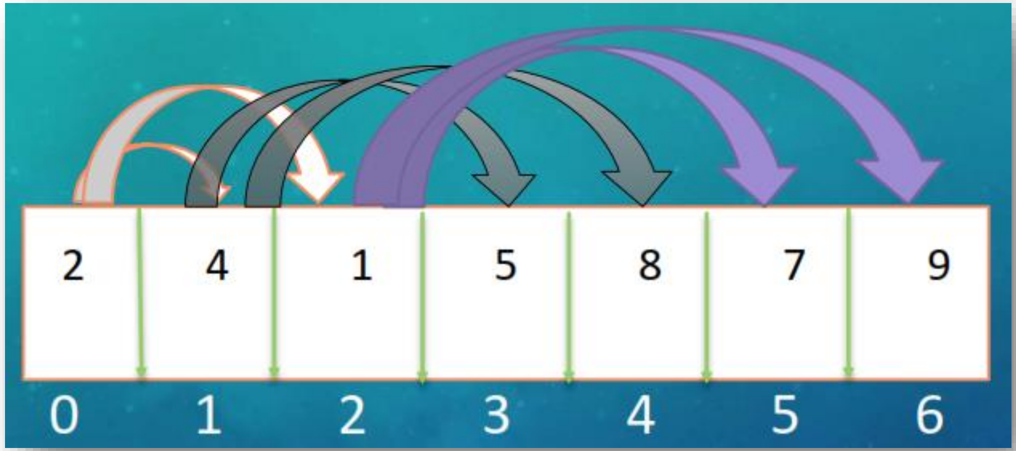
1. Dynamically created nodes

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



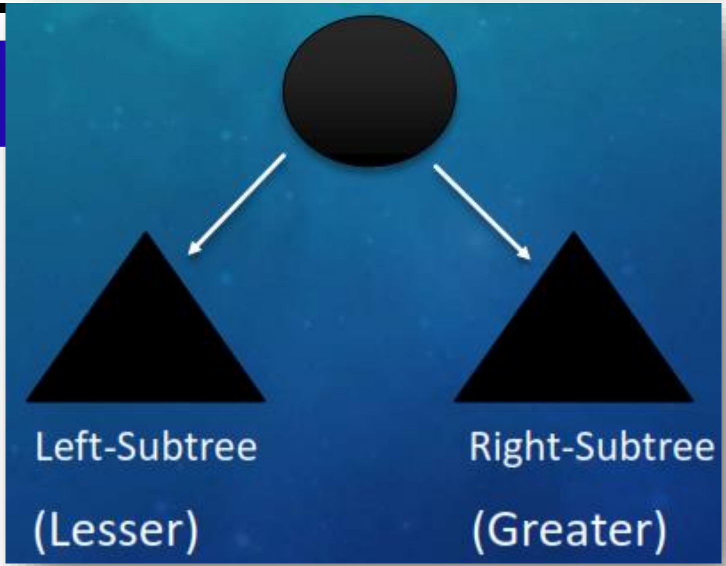
2. Array

   - It work only for Perfect Binary Tree
   - For node at index i
     - Left child's index = 2i + 1
     - Right child's index = 2i + 2

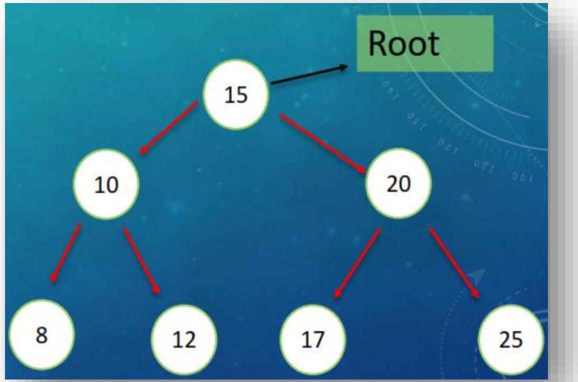


Binary Search Tree (BST)

❑ Definition



- A BST is a binary tree that is constructed in such a way that it is easy to search for the values it contains
- Rules in BST
  - ❖ All values less than (or equal) to root value are stored in the left subtree
  - ❖ All values greater than the root are stored in the right subtree
- **Example**: Suppose we have number 15 as root. We want to add **10, 20, 8, 12, 17, 25** to the tree.
  - 10<15 => 10 is inserted to left of the root
  - 20>15 => 20 is inserted to right of the root
  - 8<15 => 8 goes left
    - 8<10 => 8 goes left
  - … etc.



Insert a node in BST

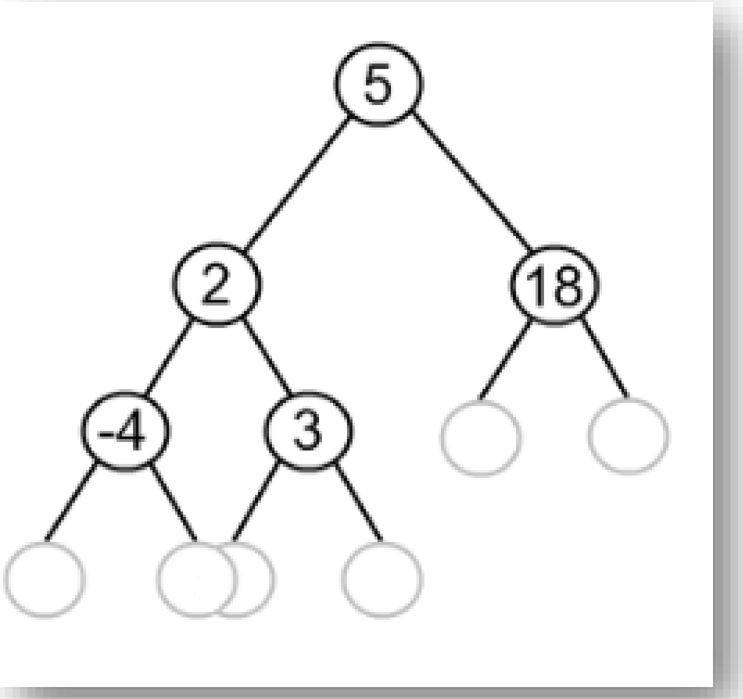
❑ Definition

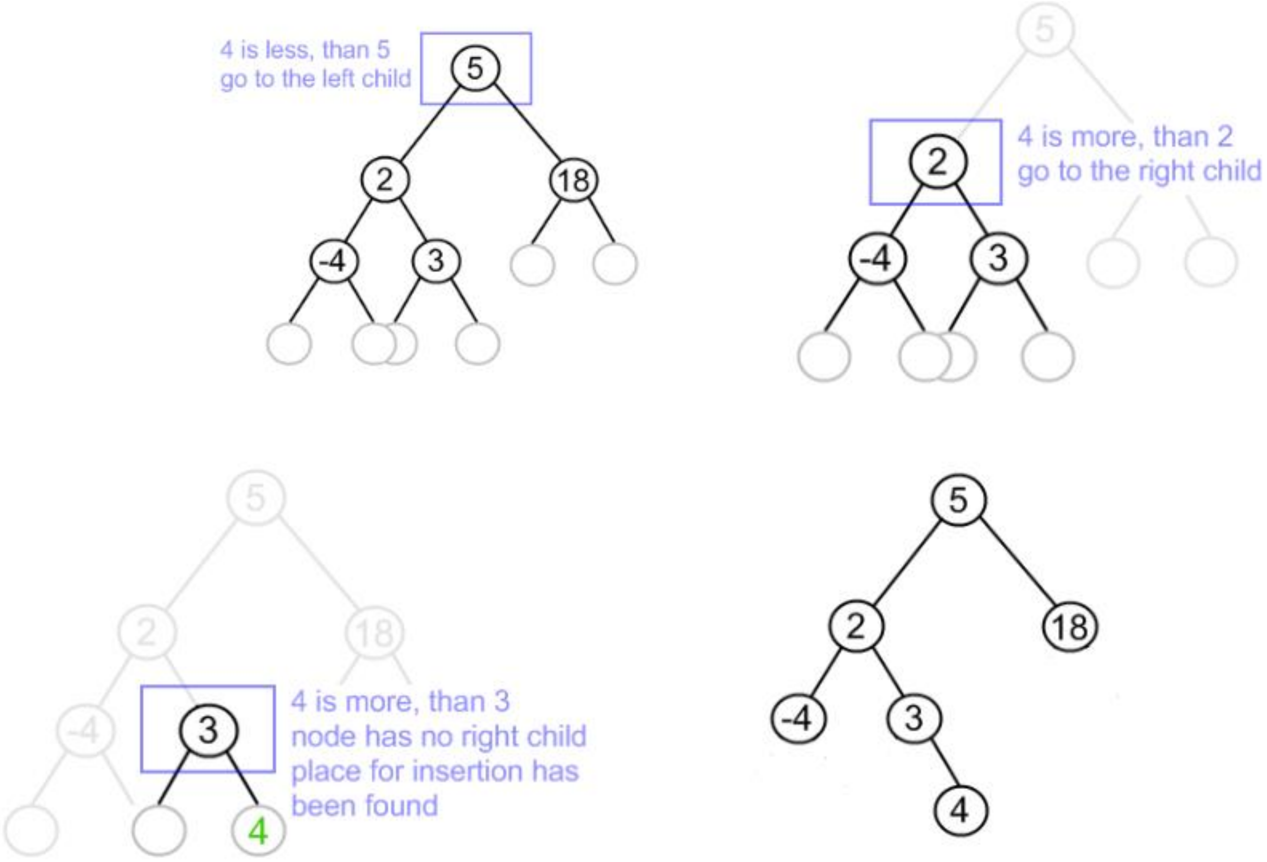
- To insert a new item in a tree, we should check that there is no duplication
  - If a new value is less than the current node's value
    - Go to the left subtree
  - Else,
    - Go to the right subtree
- **Remark**:
  - With this simple rule, the algorithm reaches a node (leaf) which has no left/right subtree
  - By the moment a place for insertion is found, we can say that a new value has no duplicate in the tree

Insert a node in BST

❑ Example

- Given a tree below on the left. How to add node of 4 to this tree?





Insert data to a tree by knowing the tree’s root

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

Insert data

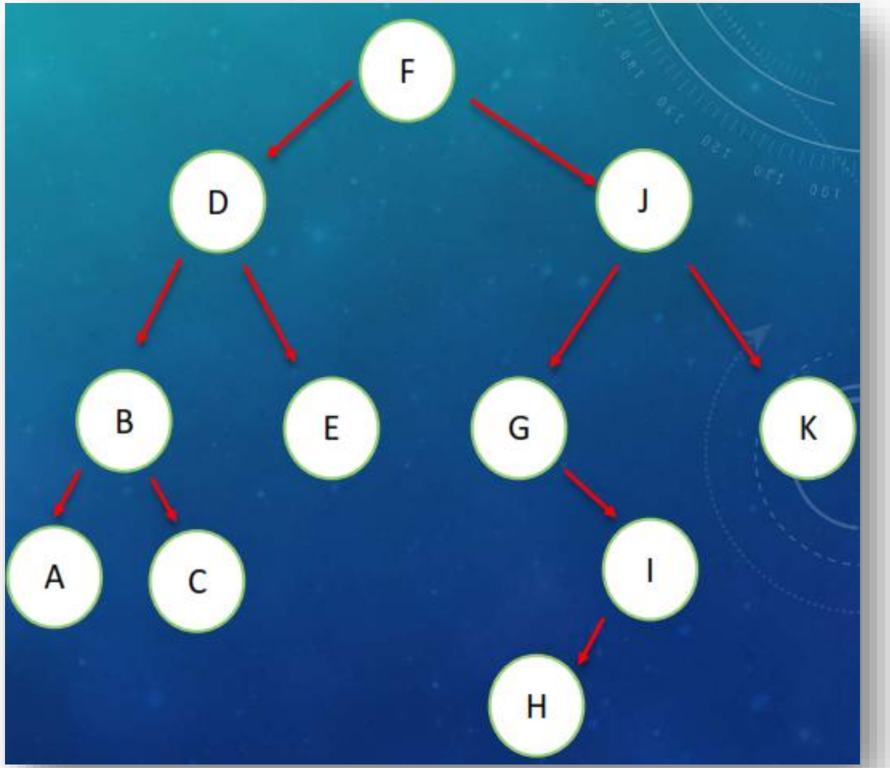
Go left

Go right

Traversal of Binary Tree

❑ Definition

- Traversal of a tree is a way that is used to visit each node in the tree
- 2 main types of tree traversal
  1. Breadth-first (level-order) traversal
     - F D J B E G K A C I H
  2. Depth-first traversal
     - Pre-order
     - **In-order**
     - Post-order



Depth-first Traversal

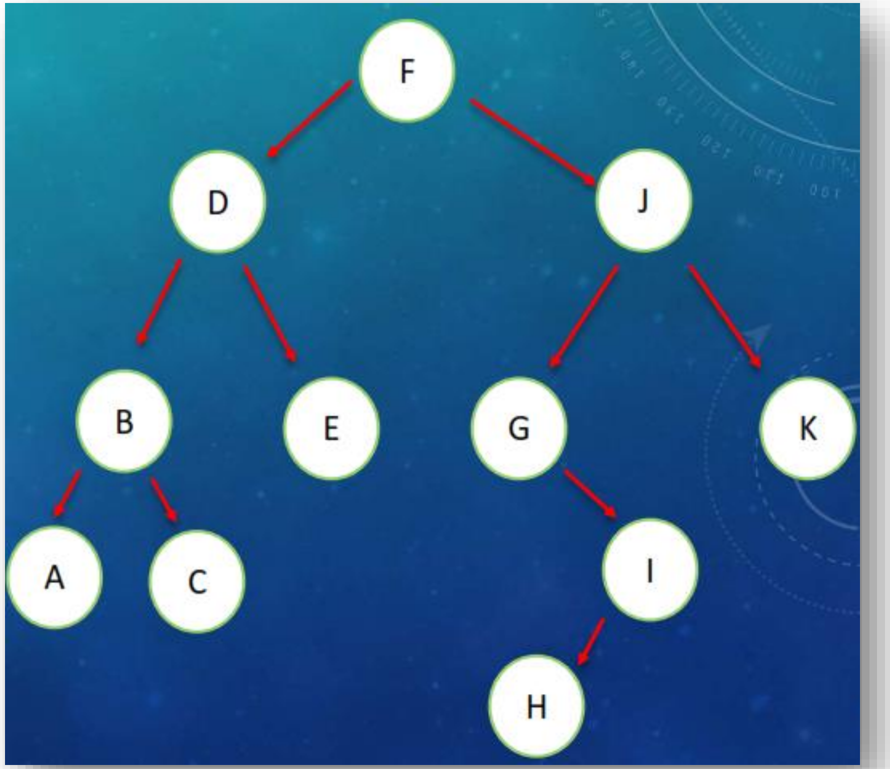
1. Pre-order
2. In-order
3. Post-order

Pre-order Traversal

❑ Definition

- It follows **data-left-right order (DLR)**
- <root's data><left><right>
  - F D B A C E J G I H K

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

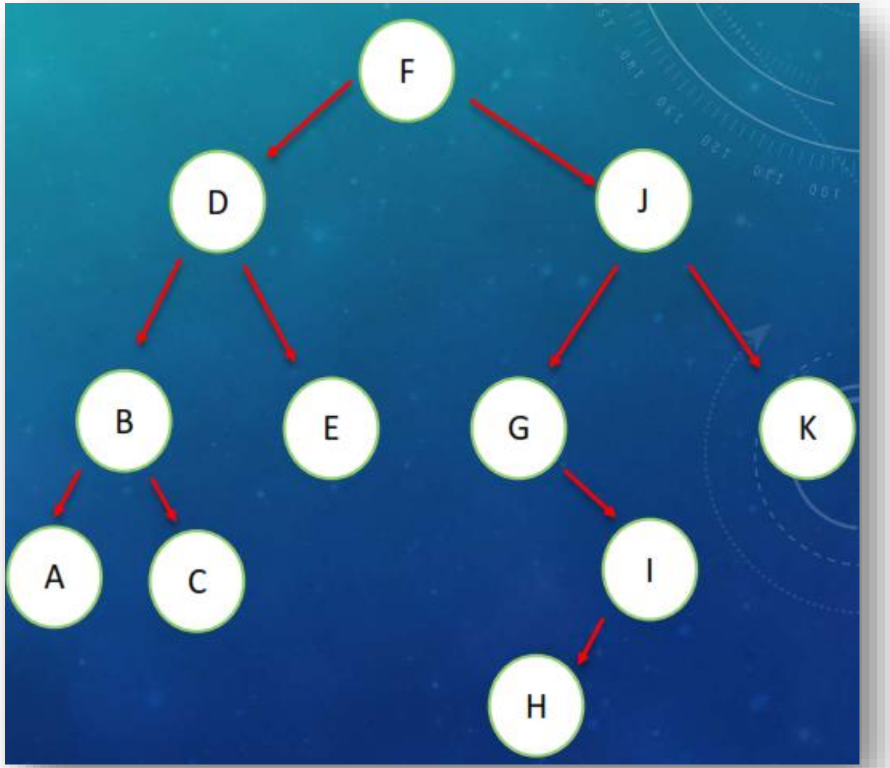


In-order Traversal

❑ Definition

- It follows **left-data-right order (LDR)**
- <left>< root's data ><right>
  - A B C D E F **G** H I J K

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

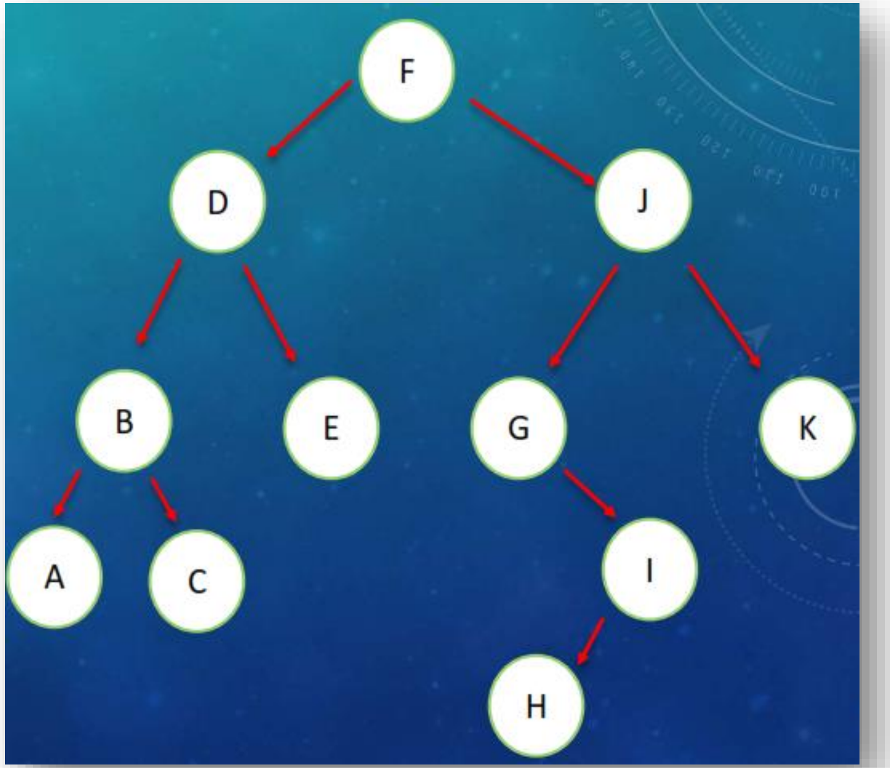


Post-order Traversal

❑ Definition

- It follows **left-right-data order (LRD)**
- <left><right><root’s data>
  - A C B E D **H** I G K J F

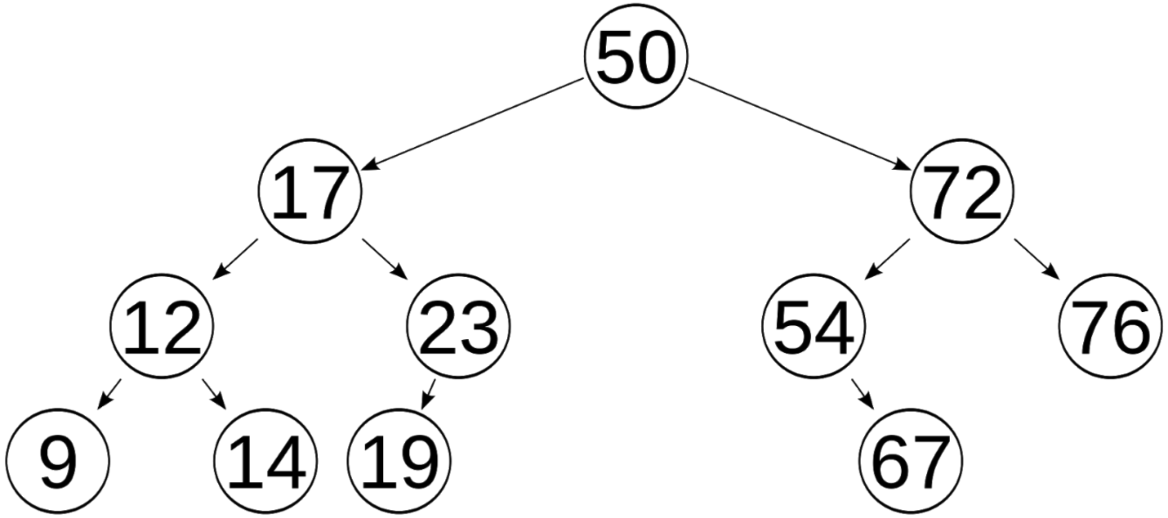
```
void postorder(Node *root){
        if(root!=NULL){
                postorder(root->left);
                postorder(root->right);
                cout<<root->data;
        }
}
```



Practice: Tree traversal

What are the outputs?

a. Pre-order traversal
b. In-order traversal
c. Post-order traversal



Outputs:

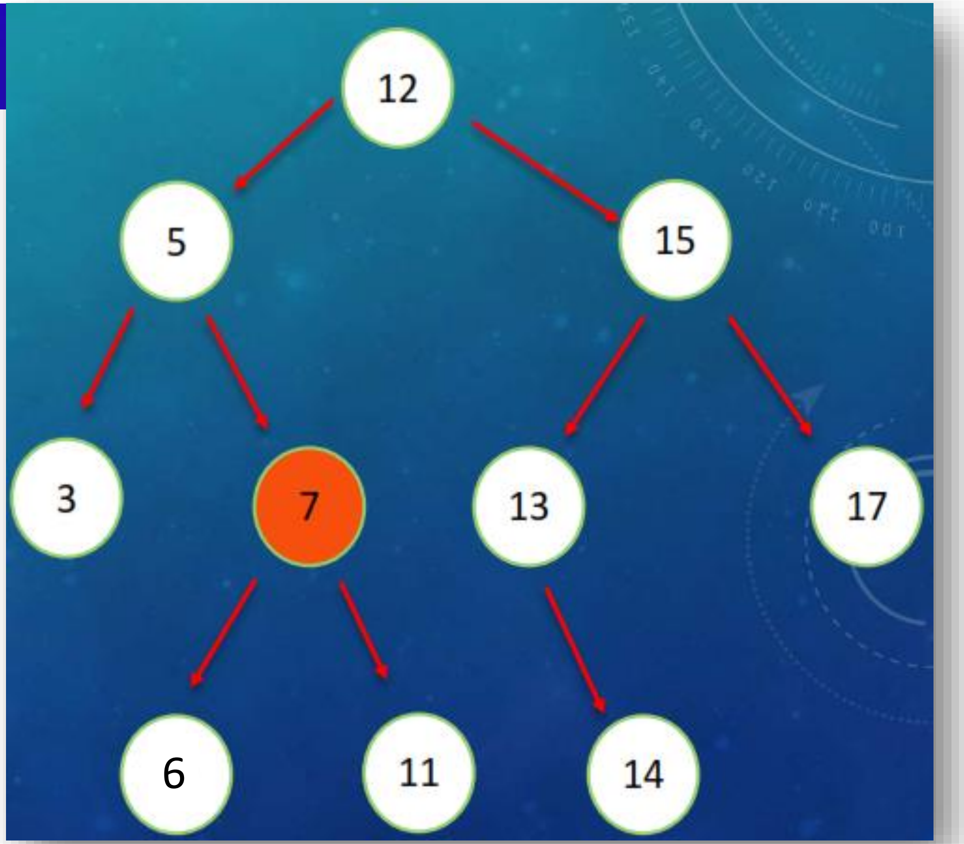
```
Pre-order traversal: 50 17 12 9 14 23 19 72 54 67 76
In-order traversal: 9 12 14 17 19 23 50 54 67 72 76
Post-order traversal: 9 14 12 19 23 17 67 54 76 72 50
```

Search for an Element in BST

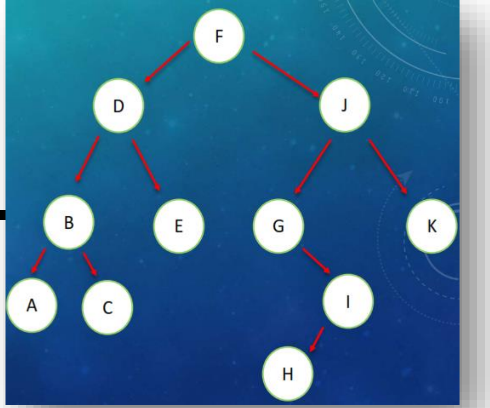
❑ Definition

- Loop to each node and compare the data

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



Combination of codes for implementing a tree



1

```
struct Node{
        char data;
        struct Node *left;
        struct Node *right;
};
```

```
int getSize(Node *root){
    if (root == NULL){
      return 0;
    }else{
      return (1 + getSize(root->left)
+ getSize(root->right));
    }
}
```

2

```
Node *insert(Node *root, char data){
        if(root==NULL){
                root=new Node( );
                root->left = NULL;
                root->right = NULL;
                root->data=data;
        }else if(data < root->data){
                root->left = insert(root->left, data);
        }else if(data > root->data){
                root->right= insert(root->right, data);
        }
        return root;
}
```

3

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

4

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

5

```
int main(){

    Node *root=NULL;
    //root=new Node; //error, no need
    root = insert(root, 'F');
    root = insert(root, 'D');
    root = insert(root, 'J');
    root = insert(root, 'B');
    root = insert(root, 'E');
    root = insert(root, 'G');
    root = insert(root, 'K');
    root = insert(root, 'A');
    root = insert(root, 'C');
    root = insert(root, 'I');
    root = insert(root, 'H');
    preorder(root); cout<<endl;
}
```

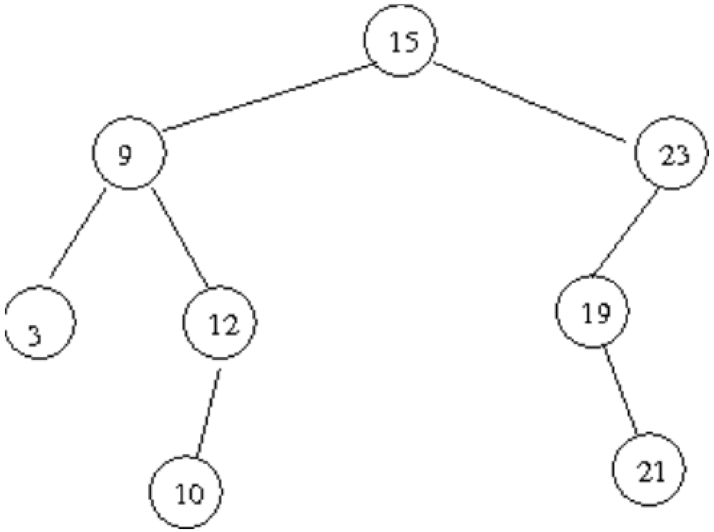
Output: F D B A C E J G I H K

In-class activity:
Now display using in-order and post-order traversal?

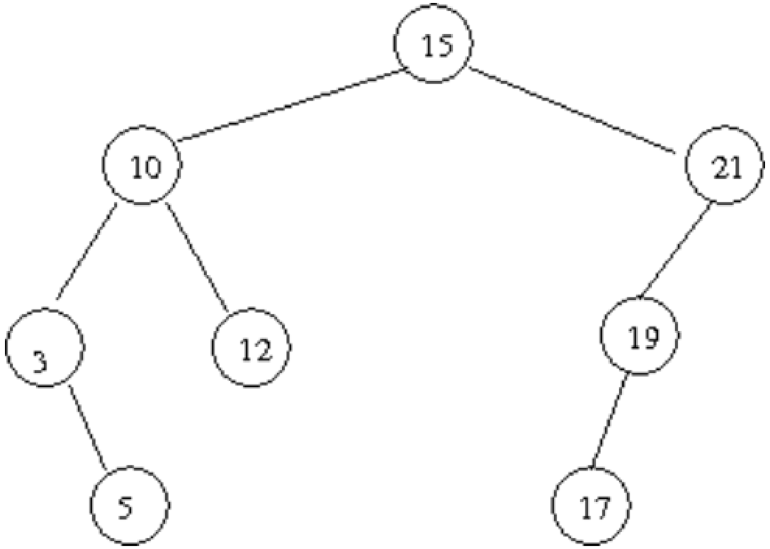
Q and A

Practices: What are the outputs of post-order, in-order, and post-order traversal for each tree?

a)



b)



Practice

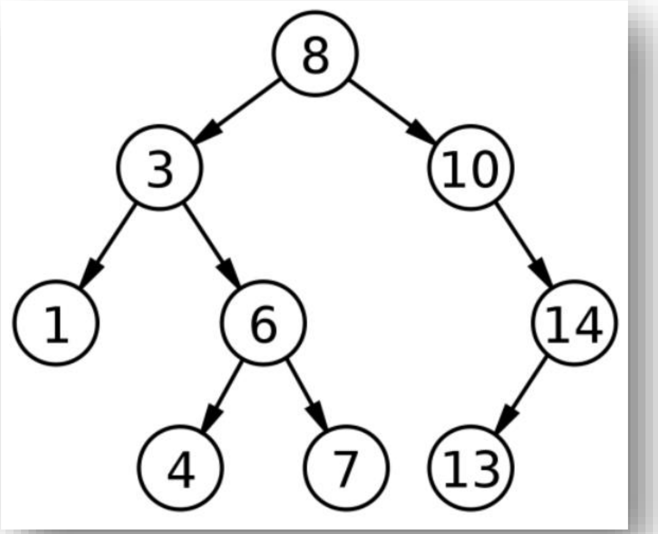
❑ Exercise

Write a program to create the tree below.

a. Display this tree using an in-order traversal.

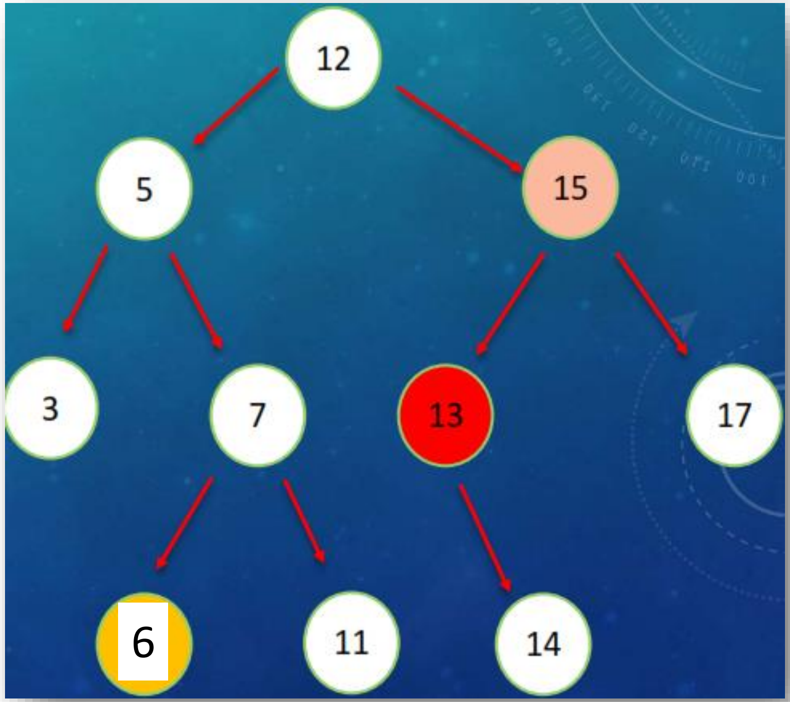
b. Keep asking a user to input a number and search whether it is in the tree.

- If exist in the tree, then display a message
  - "This number *n* is in the tree".
- Otherwise, display a message
  - "*n* does not exist in the tree"



Delete a node from BST

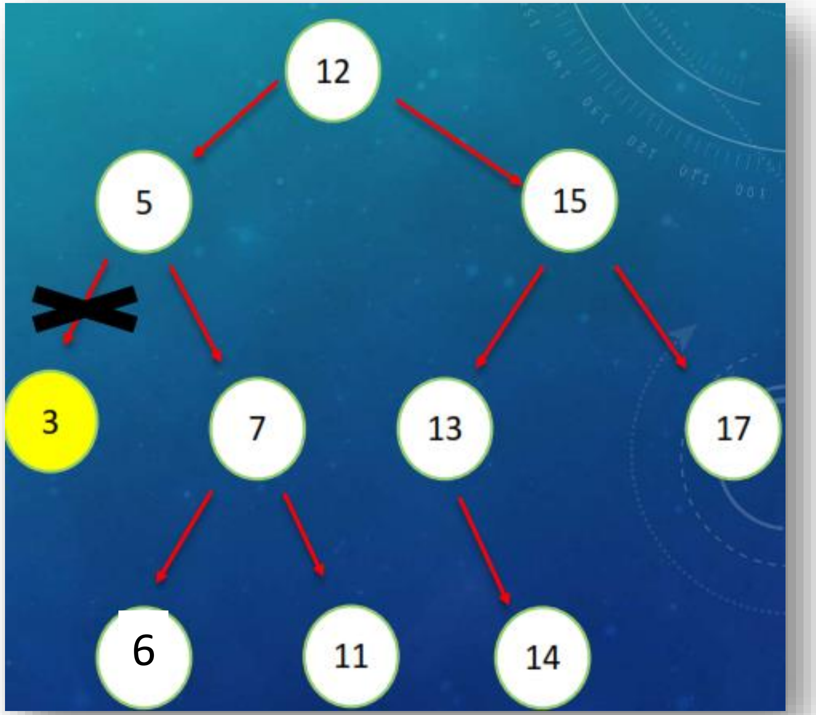
- To delete an existing node, there are 3 cases
  - Case 1: The node has no child
  - Case 2: The node has one child
  - Case 3: The node has two child



Delete a node from BST

❑ Case 1: Delete a node that has **no child**

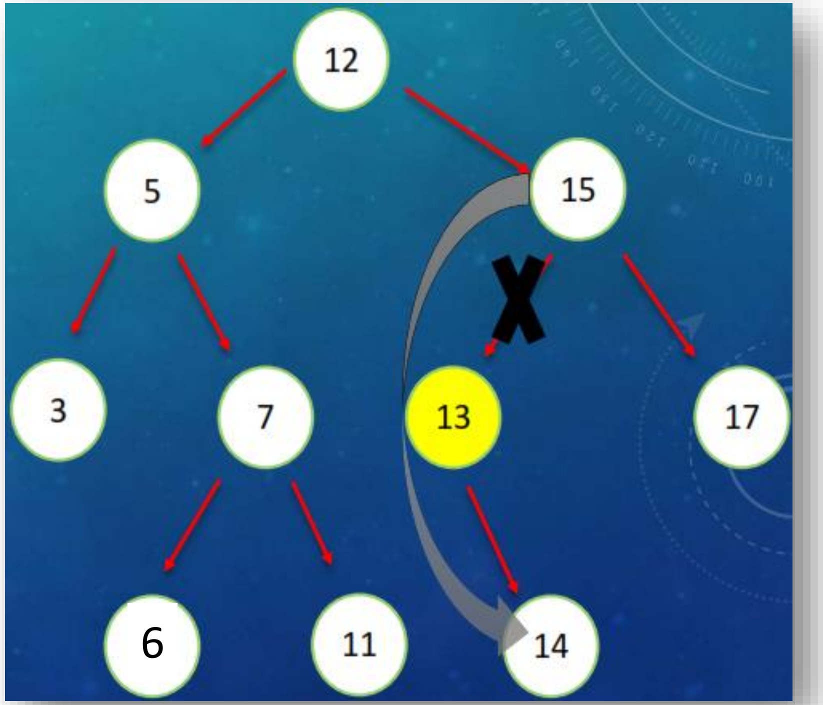
- What we need to do?
  - Make a reference to that node and its parent
    - t: the current node to be deleted
    - p: the parent of t
  - Change its parent's pointer points to NULL
    - p->left = NULL   or
    - p->right = NULL
  - Delete the node
    - delete t;



Delete a node from BST

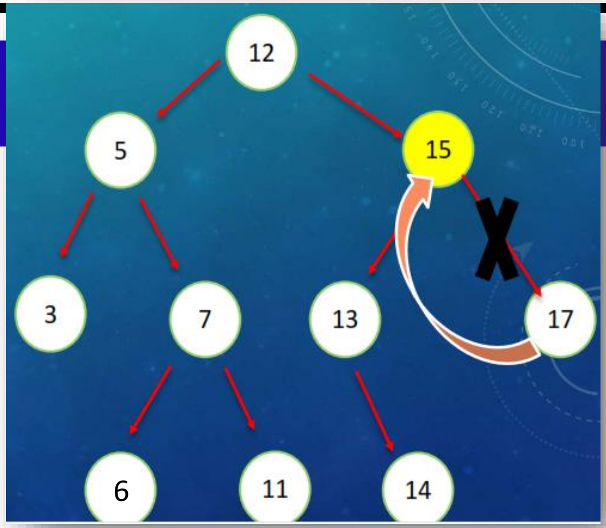
❑ Case 2: Delete a node that has **one child**

- What we need to do?
  - Make a reference to that node and its parent
    - t: the current node to be deleted
    - p: the parent of t
    - c: the child of t
  - Link its parent to its only child
    - p->left = c;   or
    - p->right = c;
  - Delete t
    - delete t;



Delete a node from BST

❑ Case 3: Delete a node that has **two children**



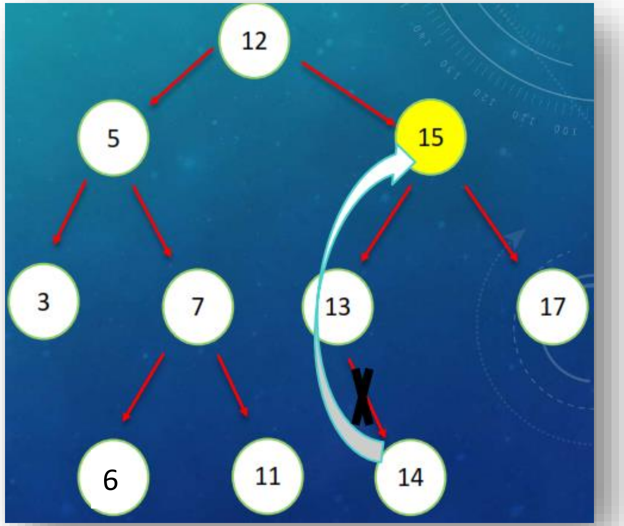
- There are 2 ways to delete

1

  - Find min in the right subtree of this node
  - Copy the value in the min node to the deleting node
  - Delete duplicate from the right subtree

2

  - Find max in the left subtree of this node
  - Copy the value in targeted node
  - Delete duplicate from the left subtree



Remark: When we delete a node that have two children, we just only

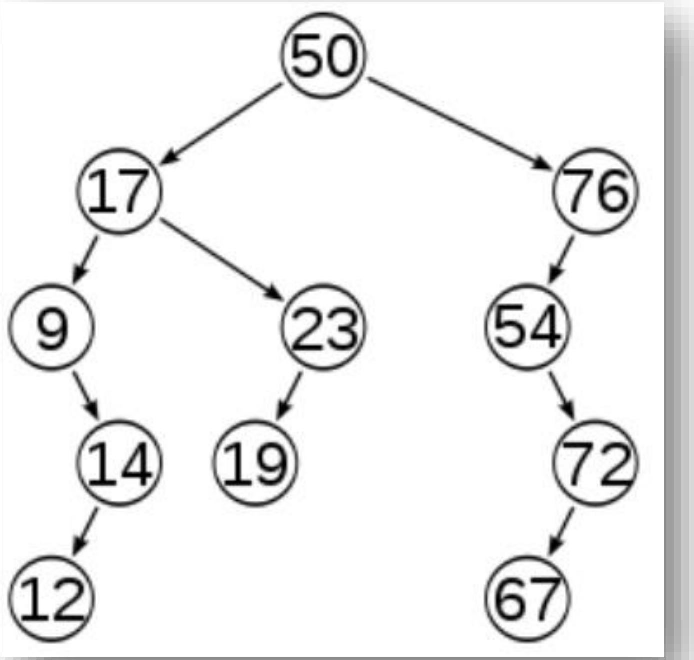
- Copy min (max) value in its right (left) subtree and put in the deleting node.
- Then delete the min (max) node since it is duplicate

Q and A

Homework

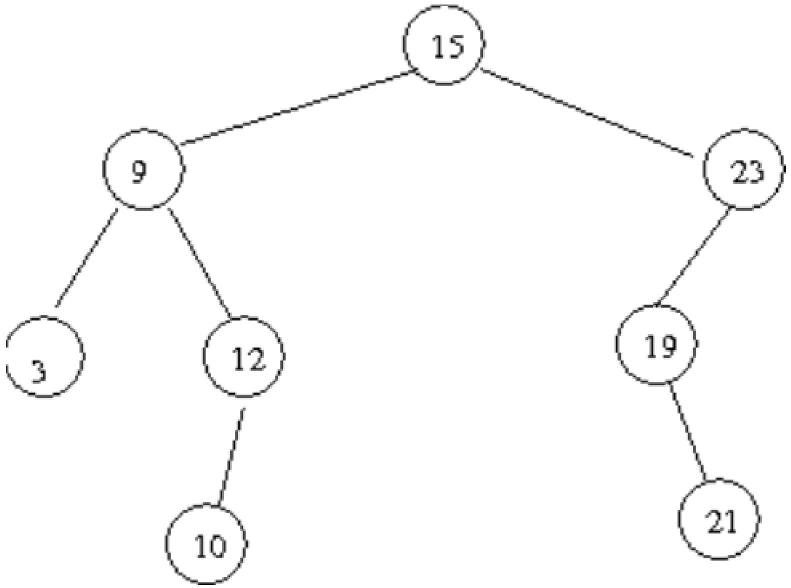
❑ Exercises

1. Write a program construct the tree as the following pictures. Write pre-order, in-order, and post-order functions to display data for each of the trees.

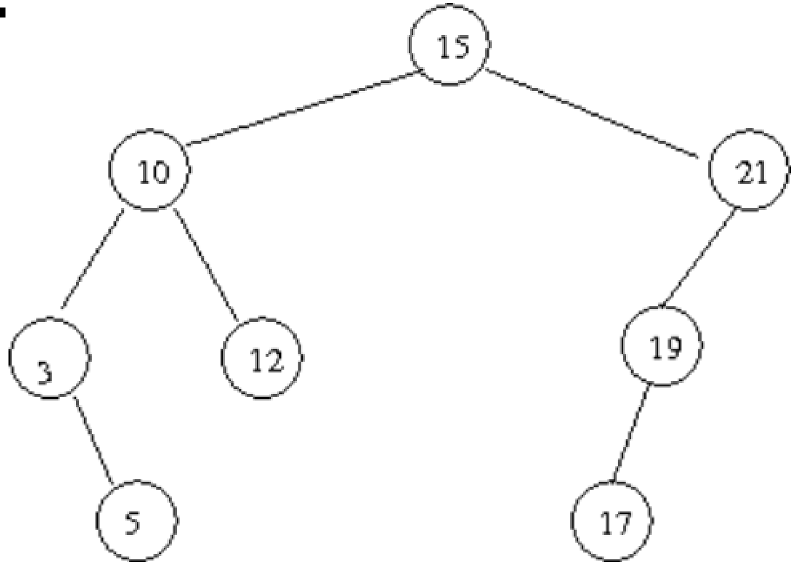


More practices: What are the outputs of post-order, in-order, and post-order traversal for each tree?

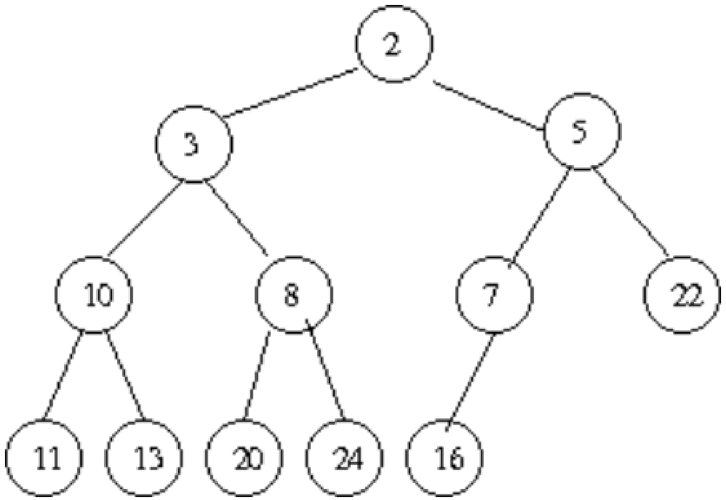
a)



b)



c)



Homework

-Hand-writing work

-Date submission: Tuesday, 18th June 19

Implementation of delete a node in BST

```
//Find min value node of a tree rooted at r
Node* findMin(Node* r){
    while(r->left != NULL) { //To find min in BST, go to the left
        r = r->left;
    }
    return r;
}
```

How to find min value node of a tree (loop to the left most)

How to find max value node of a tree?

```
// Function to delete node from a BST
void deleteNode(Node* root, int key){
    Node* parent = NULL;
    Node* curr = root;
    // Find node to be deleted and its parent node
    while (curr != NULL && curr->data != key){
        parent = curr;
        if (key < curr->data)      curr = curr->left;
        else curr = curr->right;
    }
    if (curr == NULL){
        cout<<" not found in the tree or tree is empty"; return;
    }
    //****** Case 1: node to be deleted has no children (leaf node)
    if (curr->left == NULL && curr->right == NULL){
        if (curr != root){      //if node to be deleted is not root
            if (parent->left == curr)   parent->left = NULL;
            else   parent->right = NULL;
        }else{                  // if tree has only root node
            root = NULL;
        }
        delete(curr);
    }
    //****** Case 2: node to be deleted has two children
    else if (curr->left!=NULL && curr->right !=NULL){
        Node* right  = findMin(curr->right); //find min in right subtree
        int val = right->data;
        deleteNode(root, right->data);   // recursively delete the min node
        curr->data = val;                // Copy the value to current node
    }
    //****** Case 3: node to be deleted has only one child
    else{
        Node* child=(curr->left)? curr->left: curr->right;//find child node
        if (curr != root){      //node to be deleted is not a root node
            if (curr == parent->left) parent->left = child;
            else parent->right = child;
        }else{                  //node to be deleted is root node
            root = child;
        }
        delete(curr);
    }
}
```