Data Structures Lab 8 & 9

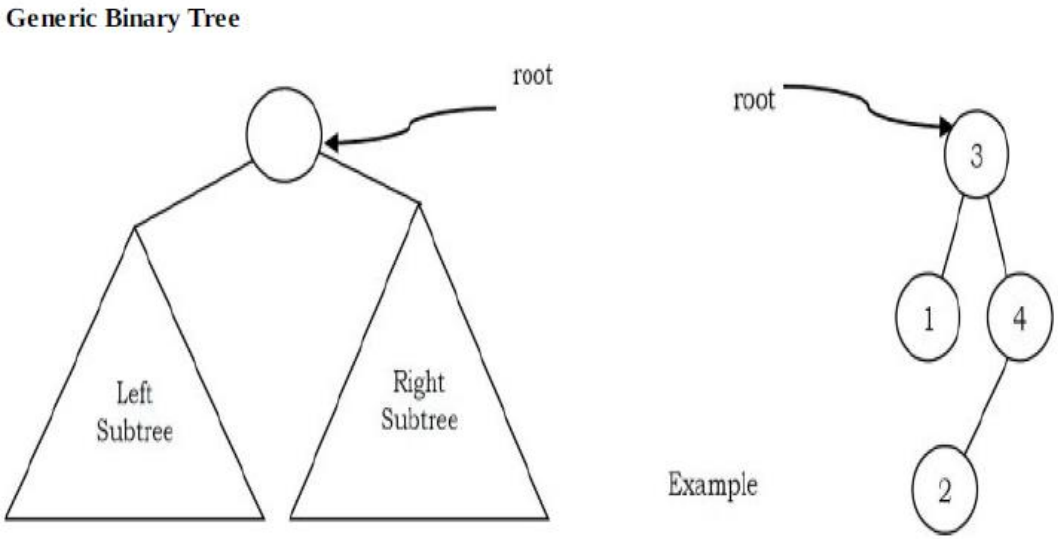
BINARY TREES

Outline

- Binary tree Introduction.
- Types of Binary tree.
- Representation of Binary tree using arrays and linked list.
- Operations of Binary tree pseudocode.
- Applications of Binary tree.
- Exercise.
- Prelab Questions.

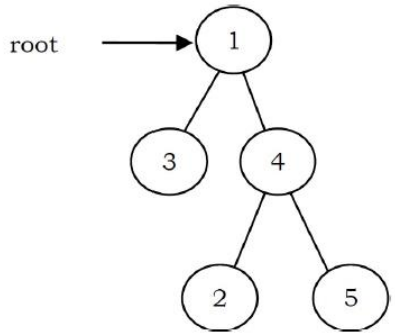
Introduction - Binary Tree

- We can represent any tree as a binary tree. Binary trees are an important type of tree structure that occurs very often.
- In a normal tree, every node can have any number of children. Binary tree is a special type of tree data structure in which every node can have a maximum of 2 children. One is known as left child and the other is known as right child.
- A tree is called **Binary Tree** if each node has zero, one or two children,
- Empty tree is also a valid binary tree.
- We can visualize a binary tree as consisting of root, and two disjoint binary trees, called the left and right sub-tree of the root.
- A Binary tree is a tree whose root has at most 2 children, each of which is a binary tree

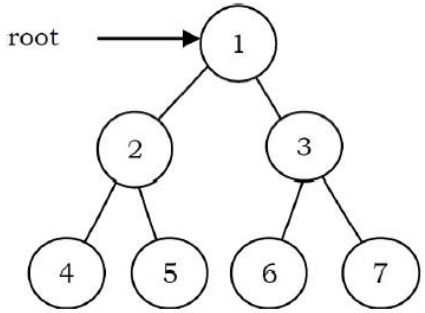


Types of Binary Trees

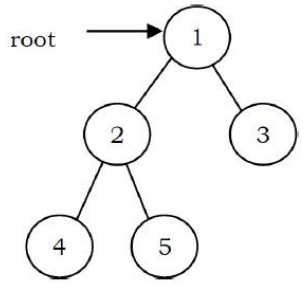
- **Strict Binary Tree:** each node has exactly two children or no children.



- **Full Binary Tree:** each node has exactly two children or no children, and all leaf nodes are at same level. (Also called as proper binary tree)



- **Complete Binary Tree:** if all leaf nodes are at level ‘h’ or ‘h-1’ and also all nodes are as far left as possible



Properties of Binary Trees

Let us assume that the height of the tree is h.

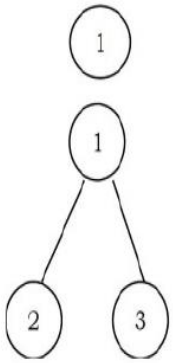
1.Number of nodes in a FULL BINARY tree is : $\mathbf{2^{h+1}-1}$

Since, there are h levels we need to add all nodes at each level: $[2^0 + 2^1 + 2^2 + \cdots + 2^h = 2^{h+1} - 1]$.

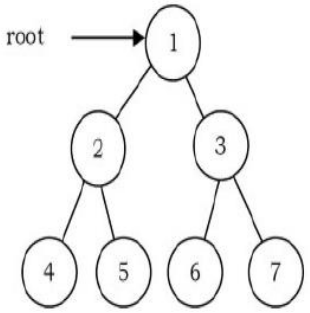
2.Height of FULL Binary tree with n nodes is : $\mathbf{\log_2(n+1)-1}$

3.Number of leaf nodes in a FULL Binary tree is : $\mathbf{2^h}$

4.No. of nodes a FULL Binary tree with L leaves is: **2L-1**



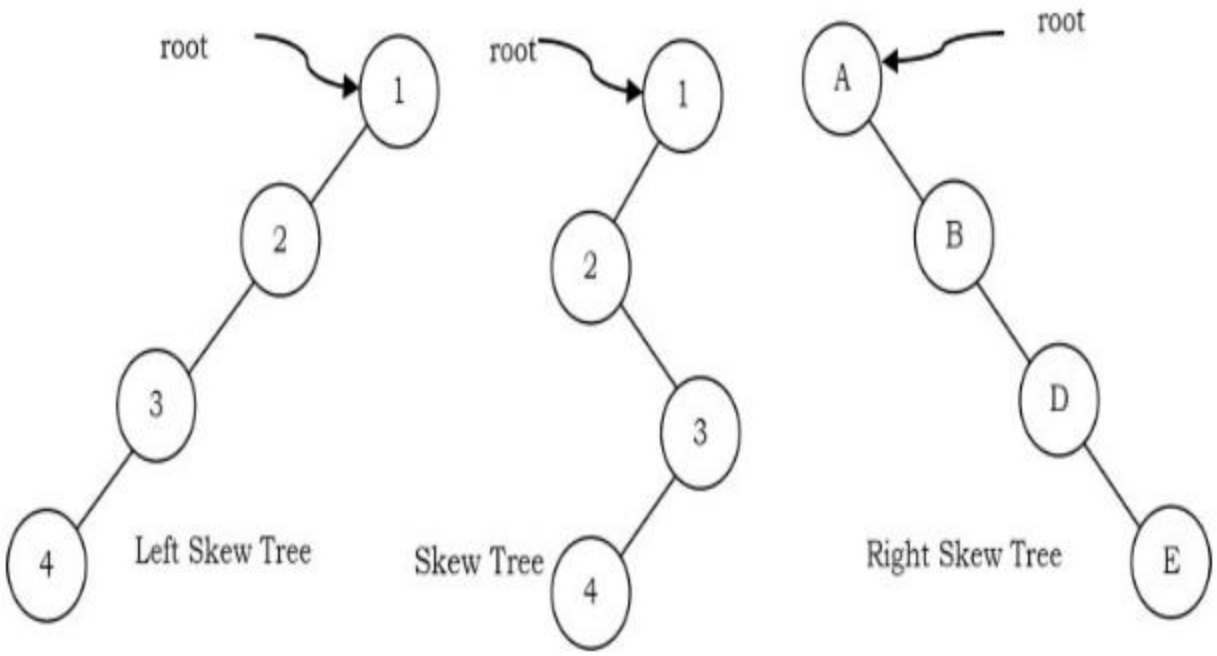
Height	Number of nodes at level h
$h = 0$	$2^0 = 1$
$h = 1$	$2^1 = 2$
$h = 2$	$2^2 = 4$



- Number of node in a complete binary tree is between : $\mathbf{2^h}$ **(min) and** $\mathbf{2^{h+1}-1}$**(max)**
- Number of NULL links (waste pointers) in a complete binary tree of **n** nodes is **n+1**.

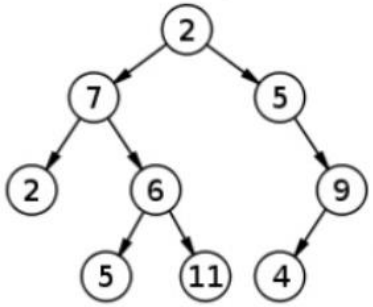
Skewed Binary Trees

If every node in a tree has only one child(except the leaf) it is called as a skewed Tree.



Representing Binary Trees : Array(Sequential) Representation

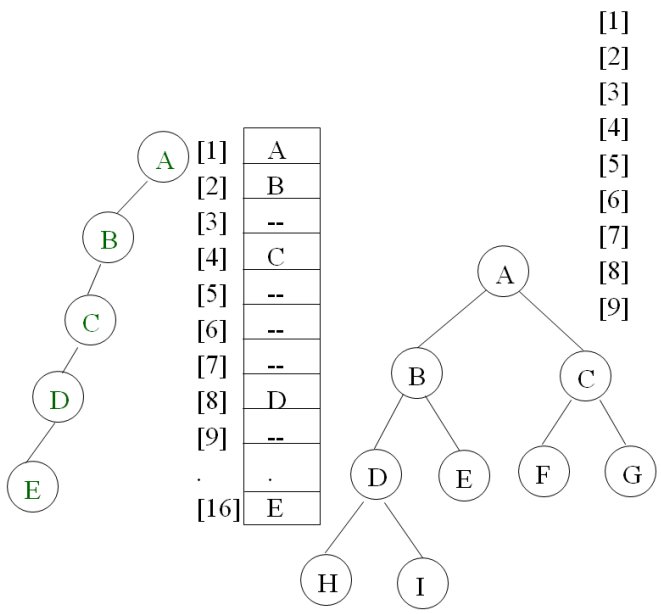
- A single array can be used to represent a binary tree.
- If the root is stored at index 0, then
  - If a node is stored at index : i
  - The left child of the node is at index : 2i+1
  - The right child of the node is at index : 2i+2



0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
2	7	5	2	6	-1	9	-1	-1	5	11	-1	-1	4	-1

- If the root is stored at index 1, then
  - If a node is stored at index : i
  - The left child of the node is at index : 2i
  - The right child of the node is at index : 2i+1

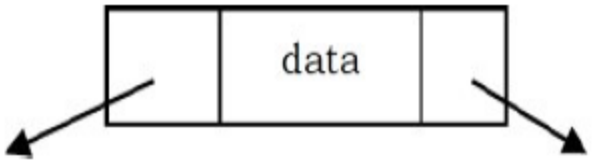
- Disadvantages :
  - Wastage of space
  - insertion/deletion problem



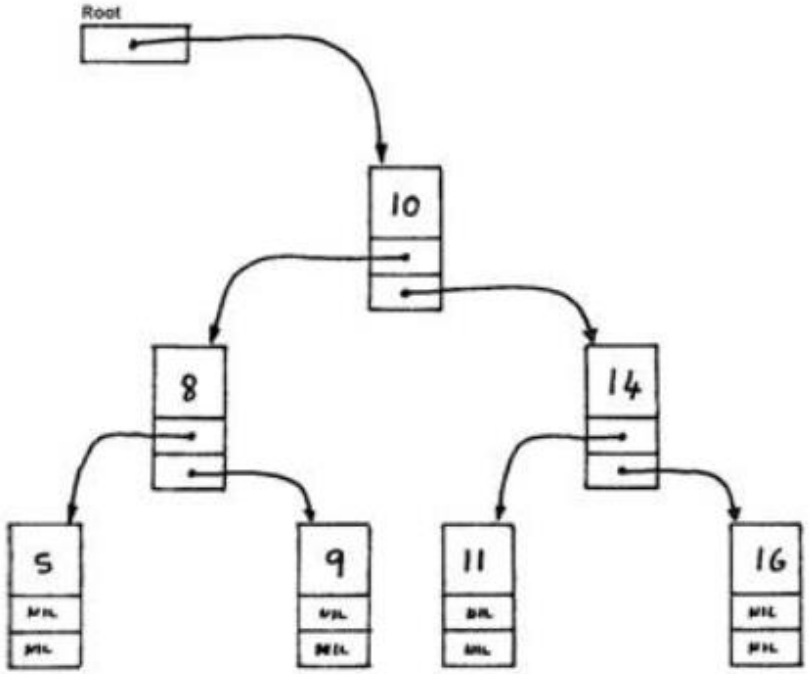
Representing Binary Trees : Linked Representation

- For simplicity, assume that the data of the nodes are integers.
- One way to represent a node (which contains data and pointers to its children along with data fields as shown below:

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



left_child right_child

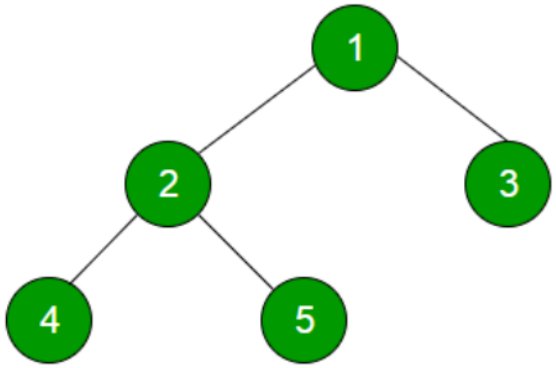


Operations on Binary Trees

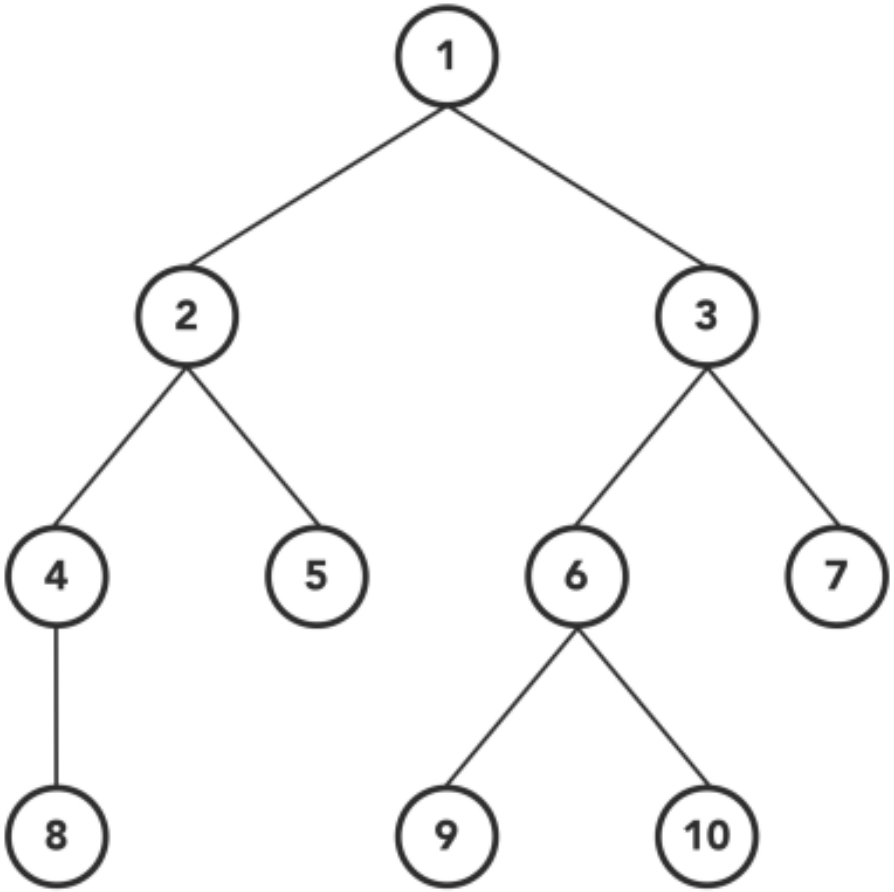
- **Basic Operations:**
  - Inserting an element into the tree
  - Deleting an element from the tree
  - Searching for an element
  - Traversing the Tree (algorithm for visiting every node in a tree)
    - **Level Order Traversal (Breadth-First Traversal)** : Starting from the root node, visit both of its children first, then all of its grandchildren, then great-grandchildren, until all the levels are covered
    - **PreOrder :** Root – Left – Right (Depth-first traversal)
    - **InOrder** : Left – Root – Right
    - **PostOrder :** Left – Right – Root
- **Auxiliary Operations:**
  - Finding the size of a node/tree
  - Finding the height/depth of the tree
  - Finding height/depth of a node
  - Finding minimum/maximum element in a binary tree
  - Finding the level which has maximum sum
  - Finding the parent/child of a given node
  - Finding ancestors/descendants of a given node
  - Finding the least common ancestor (LCA) for a given pair of nodes

Level Order Traversal – Uses Queue

- **Algorithm**
  - Visit the root
  - While traversing level l, keep all the elements at level l+1 in queue
  - Go to the next level and visit all nodes at that level
  - Repeat this until all levels are completed.



  - Level order traversal of above tree : 1 2 3 4 5


1
2
3
4
5
6
7
8
9
10

Level Order Traversal

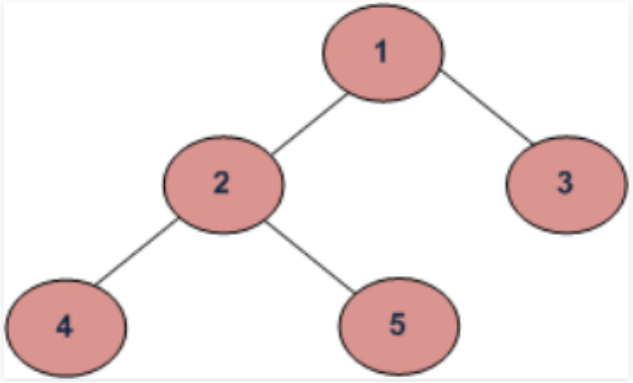
PseudoCode:

```
void LevelOrder(struct BinaryNode *root)
{
    struct BinaryNode *temp;
    struct Queue *Q= CreateQueue();
    if(!root)
          return;
    EnQueue(Q,root);
    while( !isEmpty(Q))
    {
          temp =DeQueue(Q);
          printf("%d ",temp->data);
          if(temp->left)
                    EnQueue(Q,temp->left);
        if(temp->right)
                    EnQueue(Q,temp->right);
    }
    DeleteQueue(Q);
}
```

PreOrder Traversal – Recursive(Stack)

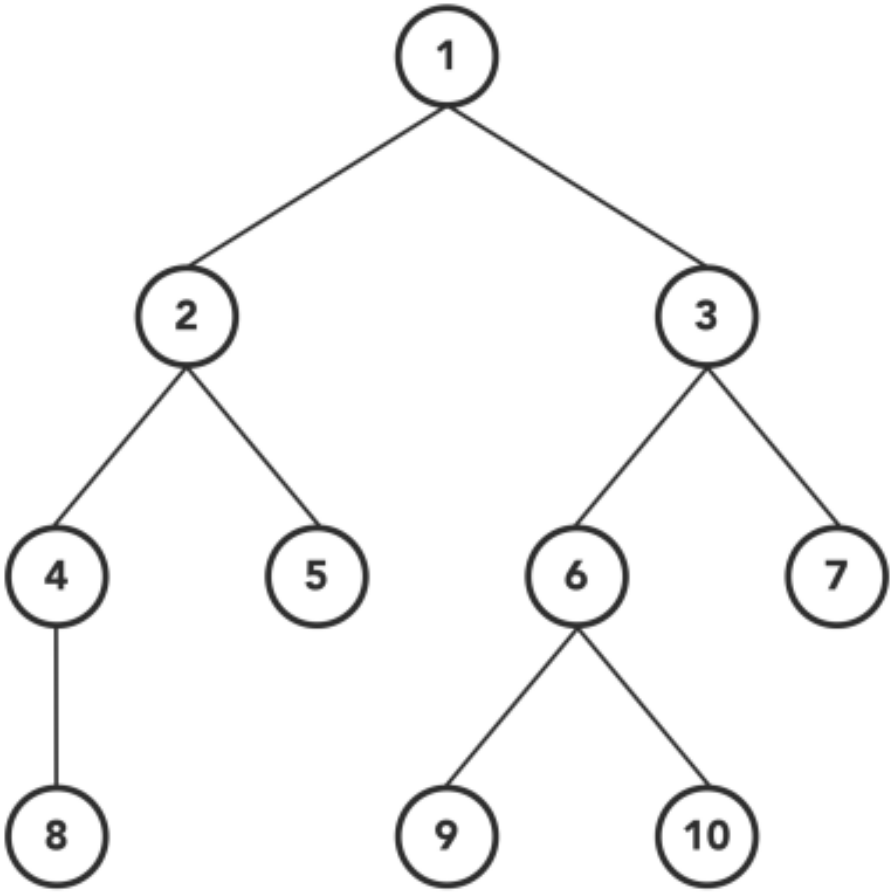
Algorithm

- Visit the root
- Traverse the left subtree in PreOrder
- Traverse the right subtree in PreOrder
- PreOrder traversal of above tree : 1 2 4 5 3



PseudoCode:

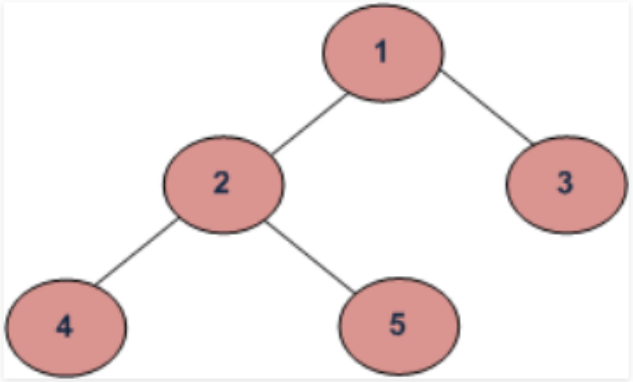
```
void PreOrder(struct BinaryNode *root)
{
    if(root)
    {
          printf("%d ",root->data);
          PreOrder(root->left);
          PreOrder(root->right);
    }
}
```


1
2
3
4
5
6
7
8
9
10

InOrder Traversal – Recursive(Stack)

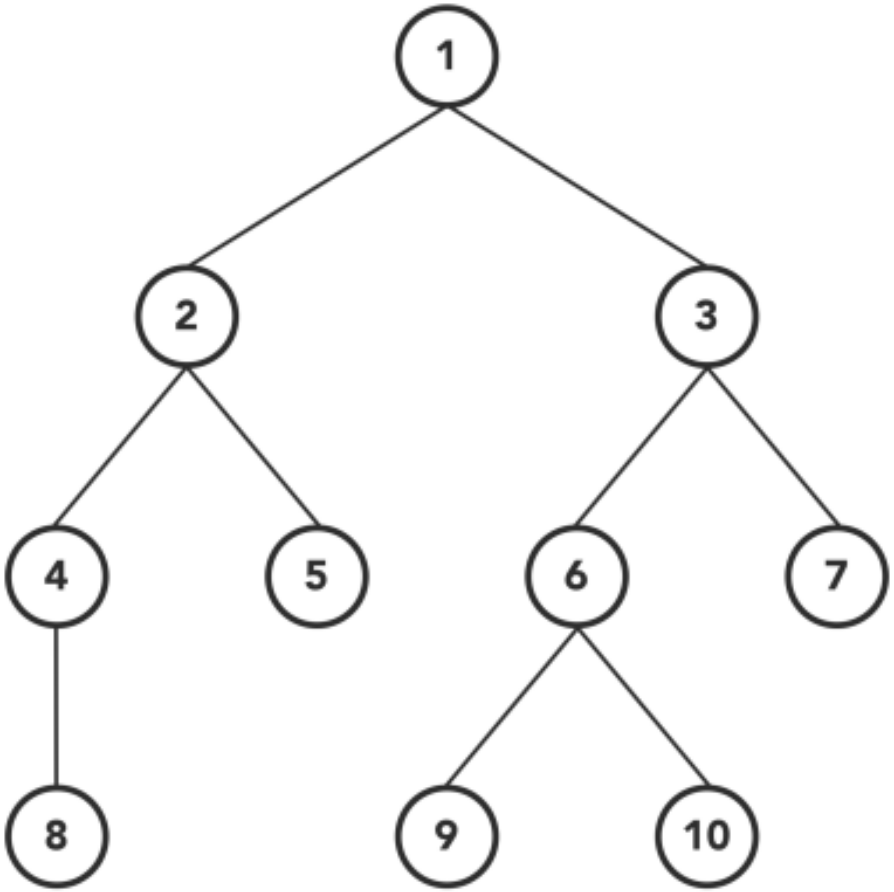
Algorithm

- Traverse the left subtree in InOrder
- Visit the root
- Traverse the right subtree in InOrder
- InOrder traversal of above tree : 4 2 5 1 3



PseudoCode:

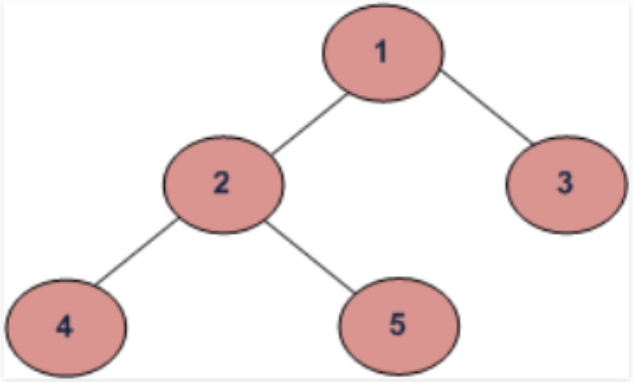
```
void InOrder(struct BinaryNode *root)
{
    if(root)
    {
          InOrder(root->left);
          printf("%d ",root->data);
          InOrder(root->right);
    }
}
```


1
2
3
4
5
6
7
8
9
10

PostOrder Traversal – Recursive(Stack)

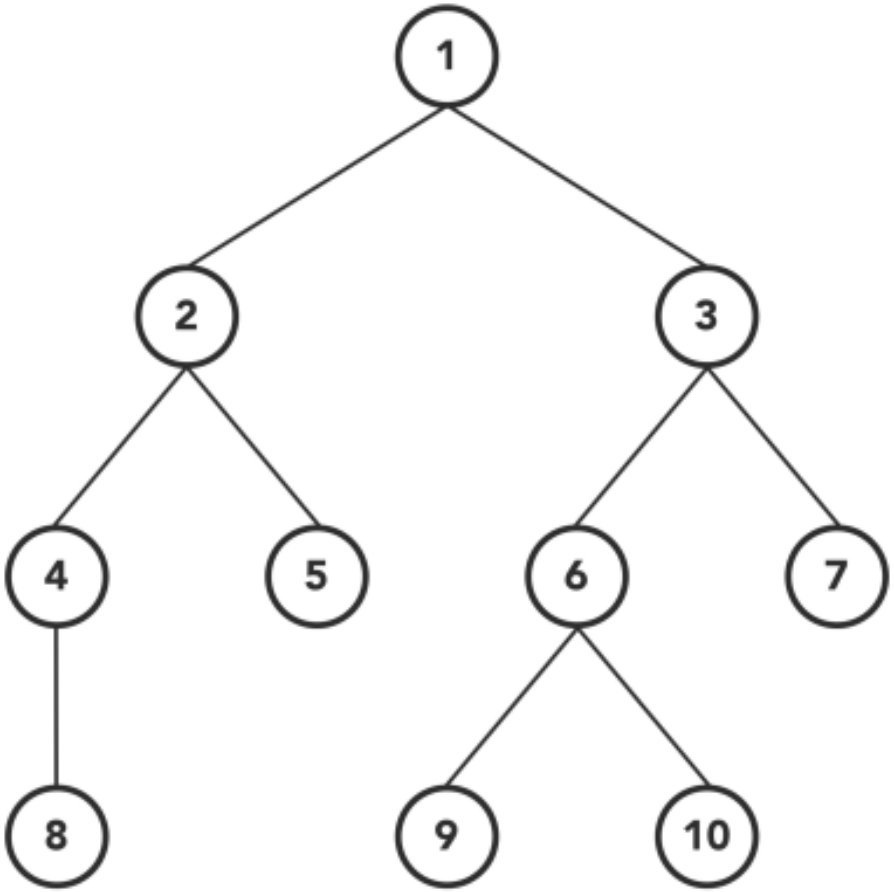
Algorithm

- Traverse the left subtree in PostOrder
- Traverse the right subtree in PostOrder
- Visit the root
- PostOrder traversal of above tree : 4 5 2 3 1



PseudoCode:

```
void PostOrder(struct BinaryNode *root)
{
    if(root)
    {
          PostOrder(root->left);
          PostOrder(root->right);
          printf("%d ",root->data);
    }
}
```


1
2
3
4
5
6
7
8
9
10

- Non-Recursive Preorder Traversal

```
void PreOrderNonRecursive(struct BinaryTreeNode *root){
    struct Stack *S = CreateStack();
    while(1) {
        while(root) {
            //Process current node
            printf("%d",root→data);
            Push(S,root);
            //If left subtree exists, add to stack
            root = root→left;
        }
        if(IsEmptyStack(S))
            break;
        root = Pop(S);
        //Indicates completion of left subtree and current node, now go to right subtree
        root = root→right;
    }
    DeleteStack(S);
}
```

Time Complexity: O(n). Space Complexity: O(n).

- Non-Recursive Inorder Traversal

```
void InOrderNonRecursive(struct BinaryTreeNode *root){
    struct Stack *S = CreateStack();
    while(1) {
        while(root) {
            Push(S,root);
            //Got left subtree and keep on adding to stack
            root = root→left;
        }
        if(IsEmptyStack(S))
            break;
        root = Pop(S);
        printf("%d", root→data);   //After popping, process the current node
        //Indicates completion of left subtree and current node, now go to right subtree
        root = root→right;
    }
    DeleteStack(S);
}
```

Time Complexity: O(n). Space Complexity: O(n).

- Non-Recursive Postorder Traversal

```
void PostOrderNonRecursive(struct BinaryTreeNode *root) {
    struct SimpleArrayStack *S = CreateStack();
    struct BinaryTreeNode *previous = NULL;
    do{
        while (root!=NULL){
            Push(S, root);
            root = root->left;
        }
        while(root == NULL && !IsEmptyStack(S)){
            root = Top(S);
            if(root->right == NULL || root->right == previous){
                printf("%d ", root->data);
                Pop(S);
                previous = root;
                root = NULL;
            }
            else
                root = root->right;
        }
    }while(!IsEmptyStack(S));
}
```

Time Complexity: O(n). Space Complexity: O(n).

Search Operation : Binary Tree

- Given a Binary Tree, return true if a node with the given data is found in the tree.

Algorithm

- Start with the root node.
- Recurse down the tree,
- Choose the left or right branch by comparing data with each node's data.

PseudoCode: With Recursion

```
int SearchKey(struct BinaryNode* root, int key)
{
    int temp;
    if(!root)
          return 0;
    else
    {
          if(data==root->data)
                    return 1;
          else
          {
                    temp = SearchKey(root->left, key);
                    if(temp!=0)
                              return temp;
                    else
                              return SearchKey(root->left, key);
          }
    }
    return 0;
}
```

Search Operation : Binary Tree

PseudoCode: Using Level Order traversal

```
int SearchKey(struct node* root, int key)
{
    struct Queue* que = createQueue(SIZE);
    Enqueue(root, que);
    while (!isEmpty(que))
    {
        struct node* temp = Dequeue(que);
         if( temp->data == key)
         {
                  return 1;
        }
        if (temp->left)
            Enqueue(temp->left, que);

        if (temp->right)
            Enqueue(temp->right, que);
    }
    return 0;
}
```

Insert Operation : Binary Tree

- Given a Binary Tree, we can insert the element wherever we want. To insert an element, we can use level order traversal and insert the element wherever we find the node whose left child or right child is NULL.

- **Algorithm**
  - If the tree is empty, initialize the root with new node
  - Else, Push root into Queue
  - Get the front node of the queue
    - If the left child of this front node doesn't exist, set the new node as the left child
    - Else if the right child of this front node doesn't exist, set the new node as the right child
  - If the front node has both left child and right child, Dequeue it
  - Enqueue the new node.

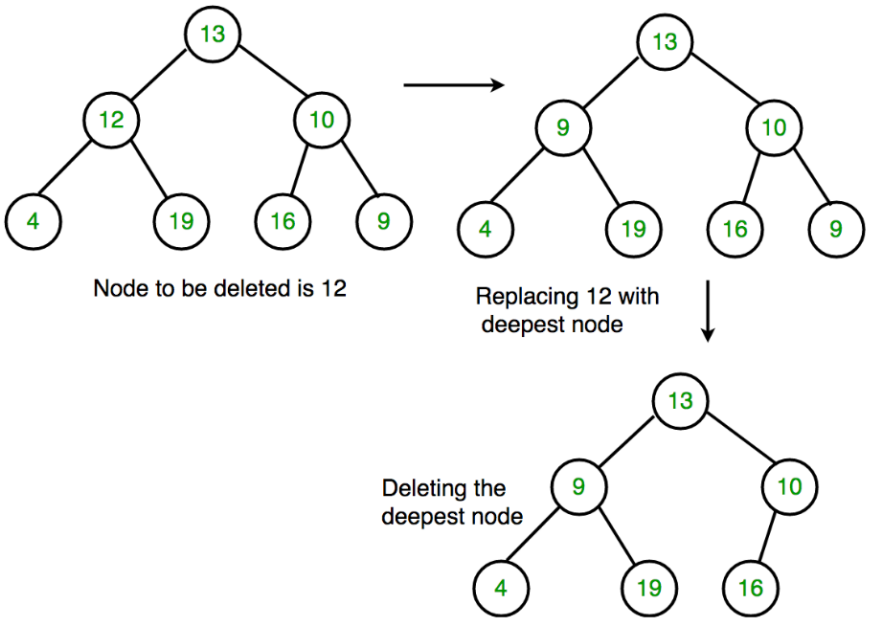
Insert Operation : Binary Tree

PseudoCode:

```
 void insert(struct BinaryNode *root, int data)
{
    struct Queue *Q;
    struct BinaryNode *temp;
    struct BinaryNode *newNode = (struct BinaryNode*) malloc(sizeof(struct BinaryNode));
    newNode->data = data;
    newNode->left = newNode->right = NULL;
    if (!root)
    {
        root = newNode;
        return;
    }
    Q=CreateQueue();
    EnQueue(Q,root);
    while(!IsEmptyQueue(Q))
    {
        temp = DeQueue(Q);
           if (temp->left)
            EnQueue(Q,temp->left);
        else
           {
           temp->left=newNode;
              DeleteQueue(Q);
              return;
        }
        if (temp->right)
            EnQueue(Q,temp->right);
        else
           {
           temp->right=newNode;
              DeleteQueue(Q);
              return;
        }
     }
    DeleteQueue(Q);
}
```

Delete Operation : Binary Tree

- Binary Tree Shrinks from the bottom
- **Algorithm**
  - Starting at root, find the node which we want to delete
  - Find the deepest, right most node in the tree.
  - Replace the deepest right most node's data with node to be deleted.
  - Then delete the deepest rightmost node.



Delete Operation : Binary Tree

PseudoCode:

```
//Deleted node is replaced by deepest node - bottom most and rightmost node
struct node* delete(struct node* root, int key)
{
    struct Queue* que = createQueue(SIZE);
    struct node *del, *last, *temp;
    Enqueue(root, que);

    while (!isEmpty(que))
    {
        temp = Dequeue(que);
          if( temp->data == key)
                    del= temp;
          if (temp->left)
              Enqueue(temp->left, que);
           if (temp->right)
              Enqueue(temp->right, que);
    }
    if (del == temp)                    // if the node to be deleted is the deepest node
          root=del_node(root, temp);
    else                                // if the node to be deleted is not the deepest node
    {
        del->data=temp->data;    //copy data of the deepest node into the node whose key has to be deleted
        root=del_node(root, temp);
    }
    return root;
}
```

Delete Operation : Binary Tree

PseudoCode- continuation:

```
struct node* del_node(struct node* root, struct node* del_node) {
    struct Queue* que = createQueue(SIZE);
    Enqueue(root, que);
     while (!isEmpty(que)) {
       struct node* temp = Dequeue(que);
       if (temp==del_node){    //deleting the last remaining node of the tree
            root=NULL;  free(temp);   return NULL;
       }
       if (temp->right){
            if(temp->right == del_node){
                   temp->right=NULL;  free(del_node); return root;
           }
             else
                   Enqueue(temp->right, que);
       }
        if (temp->left){
            if(temp->left == del_node)  {
                      temp->left=NULL; free(del_node); return root;
                   }
             else
                   Enqueue(temp->left, que);
       }
    }
}
```

Applications of Binary Trees

- Expression trees are used in compilers.
- Huffman coding trees that are used in data compression algorithms.
- Binary Search Tree (BST), which supports search, insertion and deletion on a collection of items
- Priority Queue (PQ), which supports insert, search and deletion of minimum/maximum in a collection of items .
  - In this kind of queue, the element to be deleted is the one with highest(or lowest priority).
  - At any time, an element with Arbitrary priority can be inserted into the queue.
  - For example and MaxPiorityQueue ADT
    - refer to page number 223-224 of Fundamentals of Data Structures in C, Ellis Horowitz and Sartaj Sahni, 2nd Edition
  - Applications: machine service

    -amount of time (min heap)

    -amount of payment (max heap)

Exercise

- Implementation of Operations on Binary Tree using Linked List.
  - A) Insert
  - B) Preorder
  - C) Inorder
  - D) Postorder
  - E) Level Order
  - F) Search a node value
  - G) Count number of nodes in Binary tree
  - H) Display height of a Binary tree
  - I) Display sum of nodes of a Binary tree
- Implement complete binary tree using array.
  - Insert a node
  - Preorder
  - Postorder
  - Level order
  - Search
  - Delete a node
- Implementation of Recursive and Iterative Traversals on Binary Trees.
  - Preorder
  - Postorder
  - Inorder

Prelab Questions

- Give the procedure to print left view of a Binary tree
- Give the procedure to determine if given two nodes are cousins of each other
- Give procedure to print all paths from root to leaf nodes in a binary tree.
- Procedure to build binary tree from its traversals.(Inorder,Preorder,Postorder).

In the following link questions look interesting, all the students try to learn all and code by yourself.

https://medium.com/techie-delight/binary-tree-interview-questions-and-practice-problems-439df7e5ea1f