ASSINGMENT NO.	**2**
TITLE	Binary Tree
PROBLEM STATEMENT /DEFINITION	Beginning with an empty binary tree, Construct binary tree by inserting the values in the order given. After constructing a binary tree perform following operations on it- • Perform in order / pre order and post order traversal • Change a tree so that the roles of the left and right pointers are swapped at every node • Find the height of tree • Copy this tree to another [operator=] • Count number of leaves, number of internal nodes. • Erase all nodes in a binary tree. (implement both recursive and non-recursive methods)
OBJECTIVE	To understand practical implementation and usage of binary tree for solving the problems.
OUTCOME	After successful completion of this assignment, students will be able to implement and use binary tree for efficient solution to problems
S/W PACKAGES AND **HARDWARE APPARATUS USED**	• (64-bit)64-BIT Fedora 17 or latest 64-bit update of equivalent Open source OS • Programming Tools (64-bit) Latest Open source update of Eclipse Programming frame work
REFERENCES	1. E. Horowitz S. Sahani, D. Mehata, "Fundamentals of data structures in C++", Galgotia Book Source, New Delhi, 1995, ISBN: 1678298 2. Sartaj Sahani, ―Data Structures, Algorithms andApplications in C++‖, Second Edition, University Press, ISBN:81-7371522 X.
INSTRUCTIONS FOR **WRITING JOURNAL**	1. Date 2. Assignment no. and title 3. Problem definition 4. Learning Objective 5. Learning Outcome 6. Software / Hardware requirement 7. Concepts related Theory 8. Algorithms/ Pseudo Code 9. Class ADT 10. Test cases 11. Conclusion/Analysis

Prerequisites:

- Basic knowledge of linked list, stack , queue and its operations along with its implementation
- Object oriented programming features

Learning Objectives:

- To understand practical implementation and usage of binary tree for solving the problems

Learning Outcomes:

- After successful completion of this assignment, students will be able to implement and use binary tree for efficient solution to problems

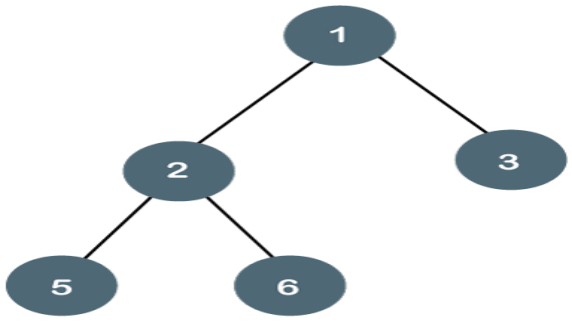
Concepts related Theory:

Binary Tree:

A tree having at-most two children is known as Binary Tree. Here, binary name itself suggests that 'two'; therefore, each node can have 0, 1 or 2 children.

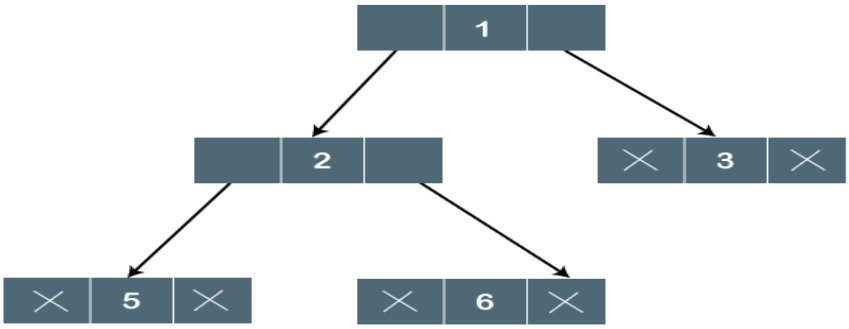
Representation:

Following is a pictorial representation of BT –



Let's understand the binary tree through an example.

The above tree is a binary tree because each node contains the at most two children. The logical representation of the above tree is given below:



In the above tree, node 1 contains two pointers, i.e., left and a right pointer pointing to the left and right node respectively. The node 2 contains both the nodes (left and right node); therefore, it has two pointers (left and right). The nodes 3, 5 and 6 are the leaf nodes, so all these nodes contain **NULL** pointer on both left and right parts.

Basic Operations:

Following are the basic operations of a tree −

- Insert − Inserts an element in a tree.
- Traversal − Perform in order / pre order and post order traversal
- Mirror − Find the mirror image of tree
- Height − Find the height of tree
- Copy − Copy this tree to another
- Count − Count number of leaves, number of internal nodes
- Deletion − Delete an element in a tree.

Properties of Binary Tree

- At each level of i, the maximum number of nodes is 2i.
- The height of the tree is defined as the longest path from the root node to the leaf node.
- The tree which is shown above has a height equal to 3. Therefore, the maximum number of nodes at height 3 is equal to (1+2+4+8) = 15. In general, the maximum number of nodes possible at height h is (20 + 21 + 22+….2h) = 2h+1 -1.
- The minimum number of nodes possible at height h is equal to h+1.

.**Subtree**: Any node in a tree and its descendants forms a sub tree.

- **Depth of a node**: the number of steps to hop from the current node to the root node of the tree.

- **Depth of a tree**: the maximum depth of any of its leaves.
- **Height of a node**: the length of the longest downward path to a leaf from that node.
- **Full binary tree**: every leaf has the same depth and every nonleaf has two children.
- **Complete binary tree**: every level except for the deepest level must contain as many nodes as possible; and at the deepest level, all the nodes are as far left as possible.

ADT:

```
class Node
  {
   int data;
   Node *left,*right

public: Node();
        Node(int);
        Friend class BTree;
 };

class BTree
{
        Node *root;

        public:
        BTree() {root=NULL;}
        //Binary Tree operations
        Insert();
        Display();
        Traversal();
        Height();
        Count();
        Copy();
        Delete();
}
```

Algorithms and Pseudo code:

Traversal:

Inorder(root)

- Traverse the left sub-tree, (recursively call inorder(root -> left).
- Visit and print the root node.
- Traverse the right sub-tree, (recursively call inorder(root -> right).

Preorder(root)

- Visit and print the root node.
- Traverse the left sub-tree, (recursively call inorder(root -> left).
- Traverse the right sub-tree, (recursively call inorder(root -> right).

Postorder(root)

- Traverse the left sub-tree, (recursively call inorder(root -> left).
- Traverse the right sub-tree, (recursively call inorder(root -> right).
- Visit and print the root node.

Pseudo code:

/* Inorder traversal of a binary tree */

```
Algorithm inorder (temp)
{
        if (temp == NULL)
                return;

        inorder(temp->left);
        print( temp->data );
        inorder(temp->right);
}
```

/* Preorder traversal of a binary tree */

```
Algorithm preorder(temp)
{
        if (temp == NULL)
                return;

        print( temp->data );
        preorder(temp->left);
        preorder(temp->right);
}
```

/* Postorder traversal of a binary tree */

```
algorithm postorder(temp)
{
        if (temp == NULL)
                return;

        print( temp->data );
        postorder(temp->left);
```

```
            postorder(temp->right);
}
```

Pseudo code for creation of Binary Tree iteratively:

```
// Function to create a new node
Node* CreateNode(data)
{
   newNode = new Node(data);
   if (!newNode) {
      print( "Memory error");
      return NULL;
   }
   return newNode;
}

// Function to insert element in binary tree
algorithm InsertNode(data)
{
   // If the tree is empty, assign new node address to root
   if (root == NULL) {
      root = CreateNode(data);
      return root;
   }

   // Else, do level order traversal until we find an empty
   // place, i.e. either left child or right child of some
   // node is pointing to NULL.
   queue<Node*> q;
   q.push(root);

   while (!q.empty()) {
      Node* temp = q.front();
      q.pop();

     print("Enter the left and right child of temp->data");
      if (temp->left != NULL)
         q.push(temp->left);
      else {
         temp->left = CreateNode(data);
      }

      if (temp->right != NULL)
         q.push(temp->right);
      else {
         temp->right = CreateNode(data);
```

```
      }
   }
}
```

Pseudo code using non recursive approach for traversal:

Algorithm inordertraversal()

{

1. set top=0, stack[top]=NULL, ptr = root
2. Repeat while ptr!=NULL
   2.1 set top=top+1
   2.2 set stack[top]=ptr
   2.3 set ptr=ptr->left
3. Set ptr=stack[top], top=top-1
4. Repeat while ptr!=NULL
   4.1 print ptr->info
   4.2 if ptr->right!=NULL then
      4.2.1set ptr=ptr->right
      4.2.2 goto step 2
   4.3 Set ptr=stack[top], top=top-1

}

Algorithm preodertraversal()

{

1. set top=0, stack[top]=NULL, ptr = root
2. Repeat while ptr!=NULL
      2.1 print ptr -> info
      2.2 if (ptr -> right != NULL)
            2.2.1 top = top +1

2.2.2 set stack [top] = ptr -> right

2.3 if (ptr -> left != NULL)

2.3.1 ptr=ptr -> left

else

2.3.1 ptr=stack[top], top=top-1

}

Algorithm postodertraversal()

{

1. set top = 0, stack [top] = NULL, ptr = root
2. Repeat while ptr!=NULL

2.1 top = top +1 , stack [top] = ptr

2.2 if (ptr -> right != NULL)

2.2.1 top = top +1

2.2.2 set stack [top] = - (ptr -> right)

2.3 ptr = ptr -> left

3. ptr = stack [top], top = top-1
4. Repeat while (ptr > 0)

4.1 print ptr -> info

4.2 ptr = stack [top], top = top-1

5. if (ptr < 0)

5.1 set ptr = - ptr

5.2 Go to step 2

}

Mirror image of binary tree:

Algorithm mirror(node)

```
{
   if (node == NULL)
      return;
   else {

      /* do conversion to the left and right subtrees */
      mirror(node->left);
      mirror(node->right);

      /* swap the pointers in this node */
      temp = node->left;
      node->left = node->right;
      node->right = temp;
   }
}
```

Algorithm to find height of binary tree

1. Initialize h=0
2. if the root is not NULL

   find the height of left sub-tree

   find the height of right sub-tree

   initialize maxHeight with the maximum of the heights of left and right sub-trees

   assign h=maxHeight + 1
3. Return h

Pseudo code:

```
Algorithm tree_height(Node *root)

{

 if (!root)

          return 0;

else

{

          l_hight = tree_height(root->left);

          r_hight = tree_height(root->right);
```

```
        if(l_hight>=r_hight)
                return l_hight+1;
        else
                return r_hight+1;
}
```

Algorithm to get the leaf node count:

Algorithm getLeafCount(node) {

If node is NULL then return 0.

Else If left and right child nodes are NULL return 1.

Else recursively calculate leaf count of the tree using below formula.

Leaf count of a tree = Leaf count of left subtree + Leaf count of right subtree

}

Pseudo code:

```
Algorithm  getLeafCount(){
   // initializing queue for level order traversal
   queue<Node*> q;
   q.push(root);
   // initializing count variable
  count = 0;
   while(!q.empty()){
      Node* temp = q.front();
      q.pop();
      if(temp->left == NULL && temp->right == NULL)
         count++;
      if(temp->left) q.push(temp->left);
      if(temp->right) q.push(temp->right);
   }
   return count;
}
```

Algorithm to get the internal node count:

1. Create a recursive function that will count the number of non-leaf nodes in a binary tree.
2. Check If root is NULL or left of root is NULL and right of root is NULL then return 0

3. Return 1 + recursive call to this function with left pointer + recursive call to this function with right pointer
4. Print the count

Pseudo code:

```
/* Computes the number of non-leaf nodes in a tree. */
Algorithm countNonleaf()
{
   // Base cases.
   if (root == NULL || (root->left == NULL && root->right == NULL))
      return 0;

   // If root is Not NULL and its one of its
   // child is also not NULL
   return 1 + countNonleaf(root->left) +
            countNonleaf(root->right);
}
```

//Function to delete all the nodes in binary tree

```
void deleteTree(Node* node) //This function traverses tree in post order to delete each and every node of the tree

{
   if (node == NULL) return;

   /* first delete both subtrees */
   deleteTree(node->left);
   deleteTree(node->right);

   /* then delete the node */
   Print( Deleting node:", node->data );
   delete node;
}
```

Conclusion: After successfully completing this assignment, Students will understand, design and implement the binary tree nonlinear data structure along with its various operations.

Sample Questions:

1. What is binary tree? Explain its uses.
2. How do you delete a node from a binary tree?

3. How do you find the depth of a binary tree?
4. Explain the difference between Binary Tree and Binary Search Tree with an example?
5. What are advantages and disadvantages of BT?
6. Explain pre-order, in-order and post-order tree traversal?
7. What are time and space complexities of operations on BT?
8. Define binary tree. Explain its common uses.