**PUNE INSTITUTE OF COMPUTER TECHNOLOGY, PUNE - 411043 Department of Computer Engineering** 

**S.No.-27, Pune Satara Road, Dhankawadi, Pune-411043**

**Data Structures and Algorithms Laboratory**

**Batch-IV (H4)**

**Name: Rushikesh Vidye**

**Roll No.: 21471**

**Class: SE4**

**Assignment No. 1**

**Title:** Beginning with an empty binary tree, Construct binary search tree by inserting the values in the order given. After constructing a binary tree perform following operations on it:-

* Perform inorder / preorder 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.

**Software Requirement:**

a) OS : Microsoft Windows 10.

b) Browser: Google Chrome.

c) VS Code.

**Hardware Requirement:**

a) Processor: Intel Core i5-8265U.

b) Ram: 8 GB DDR4 2800Mhz.

**Theory:**

**Binary Tree Data Structure:**

A tree whose elements have at most 2 children is called a binary tree. Since each element in a binary tree can have only 2 children, we typically name them the left and right child.



A Binary Tree node contains following parts.

1. Data
2. Pointer to left child
3. Pointer to right child

The topmost node is called root of the tree. The elements that are directly under an element are called its children. The element directly above something is called its parent. For example, ‘a’ is a child of ‘f’, and ‘f’ is the parent of ‘a’. Finally, elements with no children are called leaves.

A tree whose elements have at most 2 children is called a binary tree. Since each element in a binary tree can have only 2 children, we typically name them the left and right child.

**Full Binary Tree** A Binary Tree is a full binary tree if every node has 0 or 2 children. The following are the examples of a full binary tree. We can also say a full binary tree is a binary tree in which all nodes except leaf nodes have two children.

**Complete Binary Tree:** A Binary Tree is a Complete Binary Tree if all the levels are completely filled except possibly the last level and the last level has all keys as left as possible.

**Perfect Binary Tree** A Binary tree is a Perfect Binary Tree in which all the internal nodes have two children and all leaf nodes are at the same level.

In a Perfect Binary Tree, the number of leaf nodes is the number of internal nodes plus 1

 L = I + 1 Where L = Number of leaf nodes, I = Number of internal nodes.

A Perfect Binary Tree of height h (where the height of the binary tree is the longest path from the root node to any leaf node in the tree, height of root node is 1) has 2h – 1 node.

**Balanced Binary Tree**   
A binary tree is balanced if the height of the tree is O(Log n) where n is the number of nodes. For Example, the AVL tree maintains O(Log n) height by making sure that the difference between the heights of the left and right subtrees is at most 1. Red-Black trees maintain O(Log n) height by making sure that the number of Black nodes on every root to leaf paths is the same and there are no adjacent red nodes. Balanced Binary Search trees are performance-wise good as they provide O(log n) time for search, insert and delete.

**Algorithm :**

1: AddNode :: int ele :

1. If root is null
   1. Root = newnode // head
2. Else accept L or R input from user.
3. If input is L and move->left is NULL
   1. Move->left = newnode
   2. Go to Step 8
4. Else move towards left subtree
5. If input if R and Move->right is NULL
   1. Move-> right = newnode
   2. Go to Step 8
6. Else Move toward right subtree
7. Go to step 7
8. Return Root

2: GetNonLeafCount :: Node\* root:

1. If root == NULL OR (root->left && root->right are NULL)
   1. Return 0
2. Else return 1 + Left Subtree + Right Subtree

3: Get Leaf Count :: Node\* root:

1. If root == NULL
   1. Return 0
2. IF root is leaf node
   1. Return 1
3. Else
   1. Return Leaf Count Left Subtree + Leaf Count Right subtree

4: In Order :: Node\* root:

1. Traverse Left Sub Tree
2. Visit the Root
3. Traverse the Right Sub Tree

5: Pre Order :: Node\* root:

1. Visit The Root
2. Traverse Left Subtree
3. Traverse Right Subtree

6: Post Order :: Node\* root

1. Traverse Left subtree
2. Traverse Right Subtree
3. Visit Root

**Time Complexity:**

|  |  |  |
| --- | --- | --- |
| Sr.No | Methods | Complexity |
| 1 | Insert Node | O(n) |
| 2 | Get Non Lead Node | O(n) |
| 3 | In Order , Pre Order, Post Order | O(n) |
| 4 | Get Leaf Node | O(n) |

**Conclusion:**

Hence we learnt the implementation of Binary tree and it’s properties as well as it’s traversal techniques.

**Code:**

|  |
| --- |
| #pragma once  #define \_CRT\_SECURE\_NO\_WARNINGS  #include<iostream>  #include<stdlib.h>  #include<stdio.h>  #include<conio.h>  #include<queue>  #include<stack>  using namespace std;  class Node{  private:      int data;      Node\* left, \* right;  public:      Node(int ele){          this->data = ele;          this->left = this->right = NULL;}      Node(Node\* other){          this->data = other->data;          this->left = other->left;          this->right = other->right;}      friend class BST;};  class BST{  private:      Node\* root;  public:      BST();      BST(Node\* root);      Node\* GetRoot();      void SetRoot(Node\* root);      Node\* AddNode(int ele);      int GetNonLeafCount(Node\* root);      unsigned int GetNonLeafCountIterative();      unsigned int GetLeafCount(Node\* node);      unsigned int GetLeafCountIterative();      void SwapTreeNodes(Node\* node);      void SwapTreeNodesIterative();      int TreeHeight(Node\* node);      int TreeHeightIterative();      Node\* CopyTree(Node\* root);      void DeleteTree(Node\* node);      void DeleteTreeIterative();      void Inorder(Node\* root);      void Preorder(Node\* root);      void Postorder(Node\* root);      void InorderIterative();      void PreorderIterative();      void PostorderIterative();      BST& operator = (const BST& other);};  BST::BST(){root = NULL;}  Node\* BST::GetRoot()  {return this->root;}  void BST::SetRoot(Node\* root)  {this->root = root;}  Node\* BST::AddNode(int ele){      Node\* newnode, \* move;      char ch;      newnode = new Node(ele);      if (root == NULL)          root = newnode;          printf("\n Root Created !!! ");}      else{          move = root;          while (1){              printf("\n L or R %d", move->data);              ch = \_getche();              if (ch == 'l' || ch == 'L') {                  if (move->left == NULL) {                      move->left = newnode;                      printf("\n\n\t Added At Left OF %d", move->data);                      break; }                  else                      move = move->left; }              else if (ch == 'R' || ch == 'r') {                  if (move->right == NULL) {                      move->right = newnode;                      printf("\n\n\t Added At right OF %d", move->data);                      break; }                  else                      move = move->right;}              else{                  printf("\n\n\t Invalid Input!!! ");                  break;}}}      return root;}  int BST::GetNonLeafCount(Node\* root){          if (root == NULL || (root->left == NULL &&              root->right == NULL))              return 0;          return 1 + GetNonLeafCount(root->left) + GetNonLeafCount(root->right);}  unsigned int BST::GetNonLeafCountIterative(){      if (root == NULL || (root->left == NULL &&          root->right == NULL))          return 0;      queue<Node\*> q;      int count = 0; // Initialize count of leaves      q.push(root);      while (!q.empty()){          Node\* temp = q.front();          q.pop();          if (temp->left != NULL)              q.push(temp->left);          if (temp->right != NULL)              q.push(temp->right);          if (temp->left != NULL || temp->right != NULL)              count++;}      return count;}  unsigned int BST::GetLeafCount(Node\* node){      if (node == NULL)          return 0;      if (node->left == NULL && node->right == NULL)          return 1;      else          return GetLeafCount(node->left) + GetLeafCount(node->right);}  unsigned int BST::GetLeafCountIterative(){      if (!root)          return 0;      queue<Node\*> q;      int count = 0; // Initialize count of leaves      q.push(root);      while (!q.empty()){          Node\* temp = q.front();          q.pop();          if (temp->left != NULL)              q.push(temp->left);          if (temp->right != NULL)              q.push(temp->right);          if (temp->left == NULL && temp->right == NULL)              count++;}      return count;}  void BST::SwapTreeNodes(Node\* node){      if (node == NULL)          return;      else{          Node\* temp;          SwapTreeNodes(node->left);          SwapTreeNodes(node->right);          temp = node->left;          node->left = node->right;          node->right = temp;}}  void BST::SwapTreeNodesIterative(){      if (root == NULL)          return;      queue<Node\*> q;      q.push(root);      while (!q.empty()){          Node\* curr = q.front();          q.pop();          swap(curr->left, curr->right);          if (curr->left)              q.push(curr->left);          if (curr->right)              q.push(curr->right);}}  int BST::TreeHeight(Node\* node){      if (node == NULL)          return 0;      else{          int lDepth = TreeHeight(node->left);          int rDepth = TreeHeight(node->right);          if (lDepth > rDepth)              return(lDepth + 1);          else              return(rDepth + 1);}}  int BST::TreeHeightIterative(){      if (root == NULL)          return 0;      queue<Node\*> q;      q.push(root);      int height = 0;      while (1){          int nodeCount = q.size();          if (nodeCount == 0)              return height;          height++;          while (nodeCount > 0){              Node\* node = q.front();              q.pop();              if (node->left != NULL)                  q.push(node->left);              if (node->right != NULL)                  q.push(node->right);              nodeCount--;}}}  Node\* BST::CopyTree(Node\* root){      Node\* copyRoot = NULL;      if (root != NULL){          copyRoot = new Node(root->data);          copyRoot->left = CopyTree(root->left);          copyRoot->right = CopyTree(root->right);}      return copyRoot;}  void BST::DeleteTree(Node\* node){      if (node == NULL)          return;      DeleteTree(node->left);      DeleteTree(node->right);      printf("\n Deleting node: %d", node->data);      delete node;}  void BST::DeleteTreeIterative(){      // empty tree      if (root == NULL) {          return;}      // create an empty queue and enqueue the root node      queue<Node\*> queue;      queue.push(root);      Node\* front = NULL;      while (!queue.empty()){          front = queue.front();          queue.pop();          if (front->left) {              queue.push(front->left);}          if (front->right) {              queue.push(front->right);}          printf("\n Deleting node: %d", front->data);          delete front;}      root = NULL;}  void BST::Inorder(Node\* root){      if (root != NULL){          Inorder(root->left);          printf("  %d", root->data);          Inorder(root->right);}}  void BST::Preorder(Node\* root){      if (root != NULL){          printf("  %d", root->data);          Preorder(root->left);          Preorder(root->right);}}  void BST::Postorder(Node\* root){      if (root != NULL){          Postorder(root->left);          Postorder(root->right);          printf("  %d", root->data);}}  void BST::InorderIterative(){      stack<Node\*> s;      Node\* curr = root;      while (curr != NULL || s.empty() == false) {          /\* Reach the left most Node of the             curr Node \*/          while (curr != NULL){              /\* place pointer to a tree node on                 the stack before traversing                the node's left subtree \*/              s.push(curr);              curr = curr->left;}          /\* Current must be NULL at this point \*/          curr = s.top();          s.pop();          printf("  %d", curr->data);          curr = curr->right;}}  void BST::PreorderIterative(){      if (root == NULL)          return;      stack<Node\*> nodeStack;      nodeStack.push(root);      while (nodeStack.empty() == false) {          Node\* node = nodeStack.top();          printf("%d ", node->data);          nodeStack.pop();          if (node->right)              nodeStack.push(node->right);          if (node->left)              nodeStack.push(node->left); }}  void BST::PostorderIterative(){      if (root == NULL)          return;      stack<Node\*> s1, s2;      s1.push(root);      Node\* node;      while (!s1.empty()) {          node = s1.top();          s1.pop();          s2.push(node);          if (node->left)              s1.push(node->left);          if (node->right)              s1.push(node->right);}      while (!s2.empty()) {          node = s2.top();          s2.pop();          printf(" %d", node->data);}}  BST& BST::operator=(const BST& other){      Node\* newleft = NULL;      Node\* newright = NULL;      try {          newleft = new Node(other.root->left);          newright = new Node(other.root->right);}      catch (...) {          delete newleft;          delete newright;          throw;}      root->data = other.root->data;      swap(root->left, newleft);      swap(root->right, newright);      delete newleft;      delete newright;      return \*this;} |

**Output:**

|  |
| --- |
| 1. Add  2. Dipslay  3. Display Iterative  4. Swap Each Tree Node  5. Swap Each Tree Node Iterative  6. Height Of Tree  7. Print Number Of Leaf and Non Leaf Nodes Recursive and Iterative  8. Delete All Nodes  9. Delete All Nodes Iterative  10.Copy Tree Recursive Function  11.Copy Tree using = operator  0. Exit  Enter Your Choice  1  Enter Element20  Root Created !!!  Enter Element10  L or R 20l  Added At Left OF 20  Enter Element30  L or R 20r  Added At right OF 20  Enter Element25  L or R 20r  L or R 30l  Added At Left OF 30  \*\*\*\*\*\*\* Inorder \*\*\*\*\*\*\*  10 20 25 30  \*\*\*\*\*\*\* Preorder \*\*\*\*\*\*\*\*\*\*\*  20 10 30 25  \*\*\*\*\*\*\* Postorder \*\*\*\*\*\*\*\*\*\*\*  10 25 30 20  \*\*\*\*\*\*\* Inorder Iterative\*\*\*\*\*\*\*  10 20 25 30  \*\*\*\*\*\*\* Preorder Iterative\*\*\*\*\*\*\*\*\*\*\*  20 10 30 25  \*\*\*\*\*\*\* Postorder Iterative\*\*\*\*\*\*\*\*\*\*\*  10 25 30 20  Swapped Tree Nodes:  30 25 20 10  Swapped Tree Nodes Iterative:  10 20 25 30  Height Of Tree : 3  Height Of Tree Non Recursive : 3  Recursive  Number Of Leaf Nodes Of Tree : 2  Number Of Non-Leaf Nodes Of Tree : 2  Iterative:  Number Of Leaf Nodes Of Tree : 2  Number Of Non-Leaf Nodes Of Tree : 2  Copy Tree Using Recursion 10 20 25 30  Copy Tree Using = Operator 10 20 25 30  Deleting node: 10  Deleting node: 25  Deleting node: 30  Deleting node: 20  Enter Element50  Deleting node: 50 |