**K230607 DS LAB # 8 29 OCT 2024**

**Question # 1**

**A screen shot of a computer

Description automatically generated**

**#include <iostream>**

**using namespace std;**

**class Node**

**{**

**public:**

**int val;**

**Node \*left;**

**Node \*right;**

**Node(int value)**

**{**

**val = value;**

**left = nullptr;**

**right = nullptr;**

**}**

**Node()**

**{**

**left = nullptr;**

**right = nullptr;**

**}**

**};**

**Node \*Built\_Tree(Node \*&root, int data)**

**{**

**if (root == nullptr)**

**{**

**return root = new Node(data);**

**}**

**if (data < root->val)**

**{**

**root->left = Built\_Tree(root->left, data);**

**}**

**else if (data > root->val)**

**{**

**root->right = Built\_Tree(root->right, data);**

**}**

**return root;**

**}**

**void Insertion(Node \*&root, int\* arr, int s)**

**{**

**int count = 0;**

**while (count < s)**

**{**

**Built\_Tree(root, arr[count++]);**

**}**

**}**

**void In\_Order\_Traversal(Node \*root)**

**{**

**if (root == nullptr)**

**return;**

**In\_Order\_Traversal(root->left);**

**cout << root->val << " ";**

**In\_Order\_Traversal(root->right);**

**}**

**int main()**

**{**

**Node \*root = nullptr;**

**int arr[] = {1,2,3,4,5};**

**int s = 5;**

**Insertion(root,arr,s);**

**cout << "Inorder Traversal for printing : " << endl;**

**In\_Order\_Traversal(root);**

**}**

**Question # 2**

**A screen shot of a computer

Description automatically generated**

**#include <iostream>**

**#include <cmath>**

**using namespace std;**

**class Node**

**{**

**public:**

**int val;**

**Node \*left;**

**Node \*right;**

**Node(int value)**

**{**

**val = value;**

**left = nullptr;**

**right = nullptr;**

**}**

**Node()**

**{**

**left = nullptr;**

**right = nullptr;**

**}**

**};**

**Node \*Build\_Tree(int arr[], int index, int size)**

**{**

**if (index >= size)**

**return NULL;**

**Node \*current = new Node(arr[index]);**

**current->left = Build\_Tree(arr, 2 \* index + 1, size);**

**current->right = Build\_Tree(arr, 2 \* index + 2, size);**

**return current;**

**}**

**int Count(Node \*current)**

**{**

**if (current == NULL)**

**return 0;**

**return 1 + Count(current->left) + Count(current->right);**

**}**

**Node \*ToFull(Node \*current)**

**{**

**if (current == NULL)**

**return NULL;**

**current->left = ToFull(current->left);**

**current->right = ToFull(current->right);**

**if (current->left == NULL && current->right != NULL)**

**{**

**delete current->right;**

**current->right = NULL;**

**}**

**if (current->left != NULL && current->right == NULL)**

**{**

**delete current->left;**

**current->left = NULL;**

**}**

**return current;**

**}**

**void Inorder\_Traversal(Node \*current)**

**{**

**if (current == NULL)**

**return;**

**Inorder\_Traversal(current->left);**

**cout << current->val << " ";**

**Inorder\_Traversal(current->right);**

**}**

**bool isCheckComplete(Node \*current, int index, int totalNodes)**

**{**

**if (current == NULL)**

**return true;**

**if (index >= totalNodes)**

**return false;**

**return isCheckComplete(current->left, 2 \* index + 1, totalNodes) &&**

**isCheckComplete(current->right, 2 \* index + 2, totalNodes);**

**}**

**bool isFull(Node \*current)**

**{**

**if (current == NULL)**

**return true;**

**if ((current->left == NULL && current->right) ||**

**(current->left && current->right == NULL))**

**return false;**

**return isFull(current->left) && isFull(current->right);**

**}**

**int main()**

**{**

**int arr[] = {1, 2, 3, 4, 5, 6, 7, 8};**

**int size = sizeof(arr) / sizeof(arr[0]);**

**Node \*root = Build\_Tree(arr, 0, size);**

**cout << "In-order Traversal of the Binary Tree: ";**

**Inorder\_Traversal(root);**

**int total = Count(root);**

**if (isCheckComplete(root, 0, total))**

**{**

**cout << "Complete  tree." << endl;**

**}**

**else**

**{**

**cout << "TNot a complete  tree." << endl;**

**}**

**if (isFull(root))**

**{**

**cout << "Full binary tree." << endl;**

**}**

**else**

**{**

**cout << "Not a full binary tree making  full." << endl;**

**root = ToFull(root);**

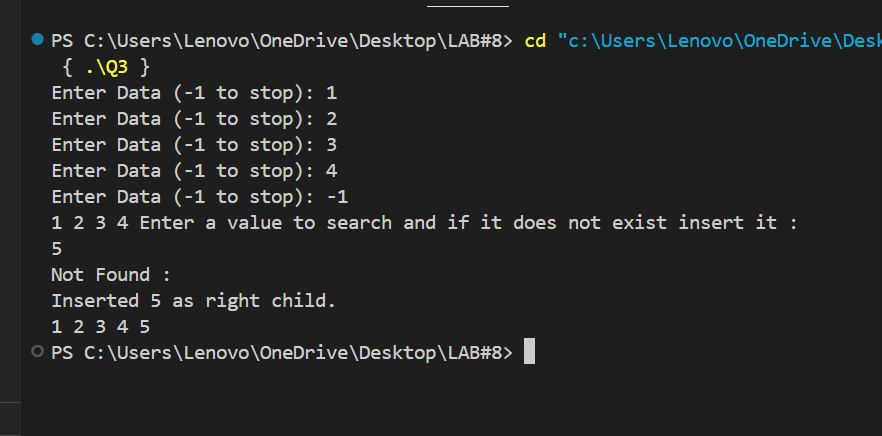
**cout << "Inorder traversal after making  tree full: ";**

**Inorder\_Traversal(root);**

**}**

**}**

**Question # 3**

****

**#include <iostream>**

**using namespace std;**

**int counter = 0;   // foe the size of tree**

**template <typename T>**

**class Queue**

**{**

**int qfront;**

**int qrear;**

**int size;**

**T \*arr;**

**public:**

**Queue(int s) : qfront(0), qrear(0), size(s)**

**{**

**arr = new T[size];**

**}**

**~Queue()**

**{**

**delete[] arr;**

**}**

**void enqueue(T val)**

**{**

**if (qrear == size)**

**{**

**cout << "Queue Overflow" << endl;**

**}**

**else**

**{**

**arr[qrear] = val;**

**qrear++;**

**}**

**}**

**void dequeue()**

**{**

**if (qrear == qfront)**

**{**

**cout << "Queue is Empty" << endl;**

**}**

**else**

**{**

**qfront++;**

**if (qfront == qrear)**

**{**

**qfront = 0;**

**qrear = 0;**

**}**

**}**

**}**

**T front()**

**{**

**if (qfront == qrear)**

**{**

**cout << "Queue is Empty" << endl;**

**return T();**

**}**

**else**

**{**

**return arr[qfront];**

**}**

**}**

**bool empty()**

**{**

**return qfront == qrear;**

**}**

**};**

**class Node**

**{**

**public:**

**int val;**

**Node \*left;**

**Node \*right;**

**Node(int value)**

**{**

**val = value;**

**left = nullptr;**

**right = nullptr;**

**}**

**Node()**

**{**

**left = nullptr;**

**right = nullptr;**

**}**

**};**

**bool Searching(Node \*root, int value)**

**{**

**if (!root)**

**return false;**

**Queue<Node \*> q(counter);**

**q.enqueue(root);**

**while (!q.empty())**

**{**

**Node \*current = q.front();**

**q.dequeue();**

**if (current->val == value)**

**{**

**return true;**

**}**

**if (current->left)**

**q.enqueue(current->left);**

**if (current->right)**

**q.enqueue(current->right);**

**}**

**return false;**

**}**

**Node \*InsertAfter(Node \*root, int value)**

**{**

**if (!root)**

**{**

**cout << "Inserted " << value << " as the root at level 0." << endl;**

**return new Node(value);**

**}**

**if (value < root->val)**

**{**

**if (!root->left)**

**{**

**root->left = new Node(value);**

**cout << "Inserted " << value << " as left child." << endl;**

**}**

**else**

**{**

**InsertAfter(root->left, value);**

**}**

**}**

**else if (value > root->val)**

**{**

**if (!root->right)**

**{**

**root->right = new Node(value);**

**cout << "Inserted " << value << " as right child." << endl;**

**}**

**else**

**{**

**InsertAfter(root->right, value);**

**}**

**}**

**else**

**{**

**cout << "Value " << value << " already exists " << endl;**

**}**

**return root;**

**}**

**Node \*Built\_Tree(Node \*&root, int data)**

**{**

**if (root == nullptr)**

**{**

**return root = new Node(data);**

**}**

**if (data < root->val)**

**{**

**root->left = Built\_Tree(root->left, data);**

**}**

**else if (data > root->val)**

**{**

**root->right = Built\_Tree(root->right, data);**

**}**

**return root;**

**}**

**void Insertion(Node \*&root)**

**{**

**double data;**

**while (true)**

**{**

**cout << "Enter Data (-1 to stop): ";**

**cin >> data;**

**if (data == -1)**

**break;**

**counter++;**

**Built\_Tree(root, data);**

**}**

**}**

**void In\_Order\_Traversal(Node \*root)**

**{**

**if (root == nullptr)**

**return;**

**In\_Order\_Traversal(root->left);**

**cout << root->val << " ";**

**In\_Order\_Traversal(root->right);**

**}**

**int main()**

**{**

**Node \*root = nullptr;**

**Insertion(root);**

**In\_Order\_Traversal(root);**

**int value;**

**cout << "Enter a value to search and if it does not exist insert it : " << endl;**

**cin >> value;**

**if (Searching(root, value))**

**{**

**cout << "Value " << value << " already exists " << endl;**

**}**

**else**

**{**

**cout << "Not Found : " << endl;**

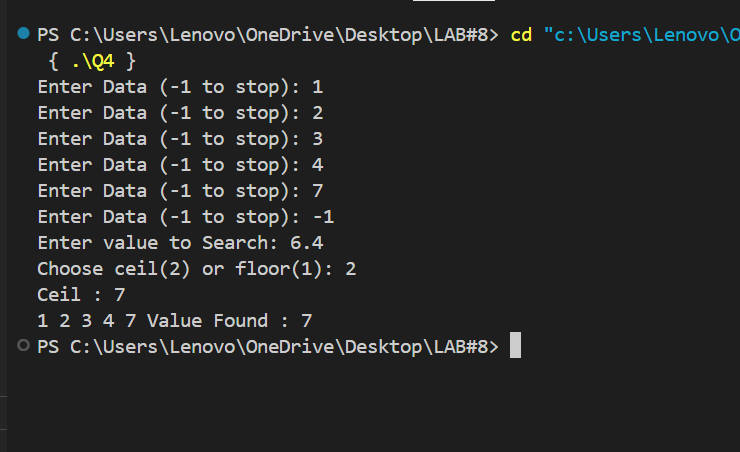
**root = InsertAfter(root, value);**

**In\_Order\_Traversal(root);**

**}**

**}**

**Question # 4**

****

**#include <iostream>**

**#include<cmath>**

**using namespace std;**

**int counter = 0; // For the size of tree**

**class Node**

**{**

**public:**

**double val;**

**Node \*left;**

**Node \*right;**

**Node(double value)**

**{**

**val = value;**

**left = nullptr;**

**right = nullptr;**

**}**

**Node()**

**{**

**left = nullptr;**

**right = nullptr;**

**}**

**};**

**template <typename T>**

**class Queue**

**{**

**int qfront;**

**int qrear;**

**int size;**

**T \*arr;**

**public:**

**Queue(int s) : qfront(0), qrear(0), size(s)**

**{**

**arr = new T[size];**

**}**

**~Queue()**

**{**

**delete[] arr;**

**}**

**void enqueue(T val)**

**{**

**if (qrear == size)**

**{**

**cout << "Queue Overflow" << endl;**

**}**

**else**

**{**

**arr[qrear] = val;**

**qrear++;**

**}**

**}**

**void dequeue()**

**{**

**if (qrear == qfront)**

**{**

**cout << "Queue is Empty" << endl;**

**}**

**else**

**{**

**qfront++;**

**if (qfront == qrear)**

**{**

**qfront = 0;**

**qrear = 0;**

**}**

**}**

**}**

**T front()**

**{**

**if (qfront == qrear)**

**{**

**cout << "Queue is Empty" << endl;**

**return T();**

**}**

**else**

**{**

**return arr[qfront];**

**}**

**}**

**bool empty()**

**{**

**return qfront == qrear;**

**}**

**};**

**void Search(Node \*root, int target, int Size)**

**{**

**Queue<Node\*> q(Size);**

**q.enqueue(root);**

**bool found = false;**

**while (!q.empty())**

**{**

**Node \*curr = q.front();**

**q.dequeue();**

**if (curr->val == target)**

**{**

**cout << "Value Found : " << curr->val << endl;**

**found = true;**

**}**

**if (curr->left)**

**q.enqueue(curr->left);**

**if (curr->right)**

**q.enqueue(curr->right);**

**}**

**if (!found)**

**cout << "Not found" << endl;**

**}**

**Node \*Built\_Tree(Node \*&root, double data)**

**{**

**if (root == nullptr)**

**{**

**return root = new Node(data);**

**}**

**if (data < root->val)**

**{**

**root->left = Built\_Tree(root->left, data);**

**}**

**else if (data > root->val)**

**{**

**root->right = Built\_Tree(root->right, data);**

**}**

**return root;**

**}**

**void Insertion(Node \*&root)**

**{**

**double data;**

**while (true)**

**{**

**cout << "Enter Data (-1 to stop): ";**

**cin >> data;**

**if (data == -1)**

**break;**

**counter++;**

**Built\_Tree(root, data);**

**}**

**}**

**void In\_Order\_Traversal(Node \*root)**

**{**

**if (root == nullptr)**

**return;**

**In\_Order\_Traversal(root->left);**

**cout << root->val << " ";**

**In\_Order\_Traversal(root->right);**

**}**

**int main()**

**{**

**Node \*root = nullptr;**

**Insertion(root);**

**float x;**

**int ch;**

**cout << "Enter value to Search: ";**

**cin >> x;**

**cout << "Choose ceil(2) or floor(1): ";**

**cin >> ch;**

**double target;**

**if (ch == 2)**

**{**

**target = ceil(x);**

**cout << "Ceil : " << target << endl;**

**}**

**else if( ch == 1)**

**{**

**target = floor(x);**

**cout << "Floor : " << target << endl;**

**}**

**else{**

**cout << "Invalid : ";**

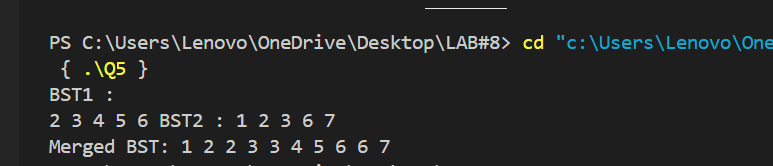
**}**

**In\_Order\_Traversal(root);**

**Search(root, target, counter);**

**}**

**Question # 5**



**#include <iostream>**

**using namespace std;**

**class Node**

**{**

**public:**

**int val;**

**Node \*left;**

**Node \*right;**

**Node(int value)**

**{**

**val = value;**

**left = nullptr;**

**right = nullptr;**

**}**

**Node()**

**{**

**left = nullptr;**

**right = nullptr;**

**}**

**};**

**void InorderTraversal(Node \*root, int arr[], int &index)**

**{**

**if (root == NULL)**

**return;**

**InorderTraversal(root->left, arr, index);**

**arr[index++] = root->val;**

**InorderTraversal(root->right, arr, index);**

**}**

**void Merge(int arr1[], int s1, int arr2[], int s2, int merge[])**

**{**

**int r = 0;**

**int c = 0;**

**int tp = 0;**

**while (r < s1 && c < s2)**

**{**

**if (arr1[r] < arr2[c])**

**merge[tp++] = arr1[r++];**

**else**

**merge[tp++] = arr2[c++];**

**}**

**while (r < s1)**

**merge[tp++] = arr1[r++];**

**while (c < s2)**

**merge[tp++] = arr2[c++];**

**}**

**Node \*Solve(int arr[], int start, int end)**

**{**

**if (start > end)**

**return NULL;**

**int mid = (start + end) / 2;**

**Node \*node = new Node(arr[mid]);**

**node->left = Solve(arr, start, mid - 1);**

**node->right = Solve(arr, mid + 1, end);**

**return node;**

**}**

**void Inorder\_Traversal(Node \*root)**

**{**

**if (root == NULL)**

**return;**

**Inorder\_Traversal(root->left);**

**cout << root->val << " ";**

**Inorder\_Traversal(root->right);**

**}**

**int main()**

**{**

**Node \*root1 = new Node(5);**

**root1->left = new Node(3);**

**root1->right = new Node(6);**

**root1->left->left = new Node(2);**

**root1->left->right = new Node(4);**

**cout << "BST1 : " << endl;**

**Inorder\_Traversal(root1);**

**Node \*root2 = new Node(2);**

**root2->left = new Node(1);**

**root2->right = new Node(3);**

**root2->right->right = new Node(7);**

**root2->right->right->left = new Node(6);**

**cout << "BST2 : ";**

**Inorder\_Traversal(root2);**

**int arr1[5];**

**int arr2[5];**

**int index1 = 0;**

**int index2 = 0;**

**InorderTraversal(root1, arr1, index1);**

**InorderTraversal(root2, arr2, index2);**

**int merge[10];**

**Merge(arr1, index1, arr2, index2, merge);**

**Node \*MergeBST = Solve(merge, 0, 9);**

**cout << endl**

**<< "Merged BST: ";**

**Inorder\_Traversal(MergeBST);**

**}**

**END**