Question #1:

#include<iostream>

#define space10

using namespace std;

template <class T>

class AVL\_Tree

{

public:

struct Node

{

T data;

Node\* left;//left adress of node

Node\* right;//right adress of node

};

Node\* root;

AVL\_Tree()

{

root = NULL;

}

int height(Node\* node)

{

if (node == NULL)//if there is no root so height will -1

{

return -1;

}

int leftTreeHeight = height(node->left);//traverse to left subtree

int rightTreeHeight = height(node->right);//traverse to right subtree

if (leftTreeHeight > rightTreeHeight)

{

return leftTreeHeight + 1;

}

else

{

return rightTreeHeight + 1;

}

}

Node rightRotation(Node root)

{

Node\* newRoot = root->left;//a temperory root node which contain the left adress of root node which has balance factor > 1

Node\* node = newRoot->right;//another temporary node which contain the right adress of first temporary node

newRoot->right = root;//the root which has balance factor >1will at root of temporary node

root->left = node;//the root which has balance factor >1 stores the adress of second temporary node at its left

return newRoot;

}

Node\* leftRotation(Node \*root)

{

Node\* newRoot = root->right;//a temperory root node which contain the right adress of root node which has balance factor < -1

Node\* node = newRoot->left;//another temporary node which contain the left adress of first temporary node

newRoot->left = root;//the root which has balance factor <-1will at root of temporary node

root->right = node;//the root which has balance factor <-1 stores the adress of second temporary node at its right

return newRoot;

}

Node \*leftRightRotation(Node \* root)

{

root->left = leftRotation(root->left);//first it will have left roatation

return rightRotation(root);//then it will have right roatation

}

Node\* rightLeftRotation(Node\* root)

{

root->right = rightRotation(root->right);//first it will have right roatation

return leftRotation(root);//then it will have left roatation

}

int BalanceFactor(Node\* node)

{

if (node == NULL)//if there is no node it will return -1

{

return -1;

}

return height(node->left) - height(node->right);//height of left subtree minus height of right subtree

}

Node\* insert(Node\* root, T data)

{

Node\* newNode = new Node;

newNode->data = data;

newNode->left = NULL;

newNode->right = NULL;

if (root == NULL)//if there is no node the first node which willl enter becomes root node

{

root = newNode;

return root;

}

if (data < root->data)//if data is less than the root's data then it will insert at left side by recursive call

{

root->left = insert(root->left, data);

}

else if (data > root->data)//if data is greater than the root's data then it will insert at right sideby recursive call

{

root->right = insert(root->right, data);

}

else

{

cout << "Duplication is not allowed in AVL tree!!" << endl;

return root;

}

int bf = BalanceFactor(root);//bf stores the balance factor which is calculate the balnace faxctor function

if (bf > 1 && data < root->left->data)//if balance factor greater than 1 and data is less thann root's left node data it has left left imbalance

{

return rightRotation(root);//as it has left left imbalance so it will perform right rotation

}

else if (bf < -1 && data > root->right->data)//if balance factor less than 1 and data is greater thann root's right node data it has right right imbalance

{

return leftRotation(root);//as it has right right imbalance so it will perform left rotation

}

else if (bf > 1 && data > root->left->data)//if balance factor greater than 1 and data is greater thann root's left node data it has left right imbalance

{

return leftRightRotation(root);

}

else if (bf < -1 && data < root->right->data)//if balance factor less than 1 and data is less than root's right node data then it has left right imbalance

{

return rightLeftRotation(root);

}

return root;

}

Node \* findMax(Node\* root)

{

Node\* temp = new Node;

temp = root;

while (temp->right != NULL)//traverse to most right position

{

temp = temp->right;

}

return temp;

}

Node\* findMin(Node\* root)

{

Node\* temp = new Node;

temp = root;

while (temp->left != NULL)//traverse to most left position

{

temp = temp->left;

}

return temp;

}

Node\* deleteNode(Node\* root, int data)

{

if (root == NULL)

{

return root;

}

if (data < root->data)//if data which u want to delete is less than root data

{

root->left = deleteNode(root->left, data);//travserse to left side and find node

}

else if (data > root->data)//if data which u want to delete is grester than root data

{

root->right = deleteNode(root->right, data);//travserse to right side and find node

}

else

{

if (root->left == NULL)//case for 1 right child

{

Node\* temp = root->right;

delete root;

return temp;

}

if (root->right == NULL)//case for 1 left child

{

Node\* temp = root->left;

delete root;

return temp;

}

else//case for two child

{

Node\* temp = findMin(root->right);

root->data = temp->data;

root->right = deleteNode(root->right, temp->data);

}

}

int bf = BalanceFactor(root);//after deleting checking balance factor if it is imbalance then performing rotation

if (bf > 1 && BalanceFactor(root->left) >= 0)

{

return rightRotation(root);

}

else if (bf > 1 && BalanceFactor(root->left) < 0)

{

return leftRightRotation(root);

}

else if (bf < -1 && BalanceFactor(root->right) <= 0)

{

return leftRotation(root);

}

else if (bf < -1 && BalanceFactor(root->right) > 0)

{

return rightLeftRotation(root);

}

return root;

}

void preorderTraversal(Node\* root)

{

if (root == NULL)

{

return;

}

cout << root->data << " ";//first print parent

preorderTraversal(root->left);//then left child

preorderTraversal(root->right);//then right child

}

void inorderTraversal(Node\* root)

{

if (root == NULL)

{

return;

}

inorderTraversal(root->left);//first print left child

cout << root->data << " ";//then parent

inorderTraversal(root->right);//then right child

}

void postorderTraversal(Node\* root)

{

if (root == NULL)

{

return;

}

postorderTraversal(root->left);//firtsly print left child

postorderTraversal(root->right);//then right

cout << root->data << " ";//then parent

}

bool searchNode(Node\* root, T data)

{

if (root == NULL)

{

return false;

}

if (data == root->data)

{

return true;

}

else if (data < root->data)//if data to search is less than the root then traversee to left to find

{

return searchNode(root->left, data);

}

else if (data > root->data)//if data to search is greater than the root then traversee to right to find

{

return searchNode(root->right, data);

}

}

int treeNodesCount(Node\* root)

{

if (root == NULL)

{

return 0;

}

int leftNodes = treeNodesCount(root->left);//node count of left subtree

int rightNodes = treeNodesCount(root->right);//node cpount of right subtree

return leftNodes + rightNodes + 1;//left subtree +right subtree +root count

}

int treeLeavesCount(Node\* root)

{

if (root == NULL)

{

return 0;

}

if (root->left == NULL && root->right == NULL)//base condition of leaves

{

return 1;

}

int leafNodes = treeLeavesCount(root->left) + treeLeavesCount(root->right);//left subtree leaves count + right subtree of tree leaves count

return leafNodes;

}

void print2d(Node\* n, int space) {//print 2d

if (n == NULL) {

return;

}

space += space;

print2d(n->right, space);

cout << endl;

for (int i = 0; i < space; i++) {

cout << " ";

}

cout << n->data << endl;

print2d(n->left, space);

}

void PrintLevelOfNOde(Node \*root, int cl)

{

if (root == NULL)

{

return;

}

else

{

cout << endl << "level of " << root->data << "is = " << cl << endl;

PrintLevelOfNOde(root->left, cl + 1);//left subtree node level

PrintLevelOfNOde(root->right, cl + 1);//prinnt right subtree node level count

}

}

};

int main()

{

int num;

int avl\_keys[10] = { 65,55,22,44,61,19,90,10,78,52 };//keys given in question for checking functions

cout << "\*- AVL FUNCTIONS \*\*\*" << endl;

AVL\_Tree<int> avl;

for (int i = 1; i < 10; i++)

{

avl.root = avl.insert(avl.root, avl\_keys[i]);

}

cout << "INORDER TRAVERSAL: ";

avl.inorderTraversal(avl.root);

cout << endl;

cout << "PREORDER TRAVERSAL: ";

avl.preorderTraversal(avl.root);

cout << endl;

cout << "POSTORDER TRAVERSAL: ";

avl.postorderTraversal(avl.root);

cout << endl;

cout << " NOW THE AVL TREE AFTER DELETION IS :" << endl;

avl.print2d(avl.root, 2);

cout << "ENTER DATA TO DELETE" << endl;

cin >> num;

avl.deleteNode(avl.root, num);

cout << "AFTER DELETING : " << num;

cout << endl;

cout << "INORDER TRAVERSAL: ";

avl.inorderTraversal(avl.root);

cout << endl;

cout << "PREORDER TRAVERSAL: ";

avl.preorderTraversal(avl.root);

cout << endl;

cout << "POSTORDER TRAVERSAL: ";

avl.postorderTraversal(avl.root);

cout << endl;

cout << " NOW THE AVL TREE AFTER DELETION IS :" << endl;

avl.print2d(avl.root, 2);

cout << "ENTER A NUMBER TO SEARCH: ";

cin >> num;

if (avl.searchNode(avl.root, num))

{

cout << "NODE IS PRESENT" << endl;

}

else

{

cout << "NODE IS NOT PRESENT" << endl;

}

cout << endl;

cout << "TOTAL NUMBER OF NODES PRESENT: " << avl.treeNodesCount(avl.root) << endl;

cout << "TOTAL NUMBER OF LEAF NODES: " << avl.treeLeavesCount(avl.root) << endl;

cout << "HEIGHT OF TREE: " << avl.height(avl.root) << endl;

cout << "LEVEL OF NODES IS: " << endl;

avl.PrintLevelOfNOde(avl.root, 0);

cout << endl;

system("pause");

return 0;

}

output:

Question #2:

#include <iostream>

#include<string>

#include <algorithm>

#include <cstring>

using namespace std;

const int MAX\_CHARS = 26;

struct CharFreq {

char ch;

int freq;

};

bool compare(CharFreq a, CharFreq b) {

return a.freq > b.freq;

}

void sortString(string& str) {

int n = str.length();

CharFreq cf[MAX\_CHARS];

memset(cf, 0, sizeof(cf));

// Count the frequency of each character

for (int i = 0; i < n; i++) {

cf[str[i] - 'a'].ch = str[i];

cf[str[i] - 'a'].freq++;

}

// Sort the characters based on their frequency

sort(cf, cf + MAX\_CHARS, compare);

// Construct the sorted string

int index = 0;

for (int i = 0; i < MAX\_CHARS; i++) {

for (int j = 0; j < cf[i].freq; j++) {

str[index++] = cf[i].ch;

}

}

}

int main() {

string str;

cout << "Enter a string: ";

getline(cin, str);

sortString(str);

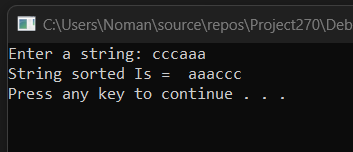
cout << "String sorted Is = " << str << endl;

system("pause");

return 0;

}

Output:



Question #3:

#include <iostream>

using namespace std;

void maxHeapify(int arr[], int n, int i)

{

int largest = i; // Initialize largest as root

int left = 2 \* i + 1; // left child

int right = 2 \* i + 2; // right child

// If left child is larger than root

if (left < n && arr[left] > arr[largest])

{

largest = left;

}

// If right child is larger than largest so far

if (right < n && arr[right] > arr[largest])

{

largest = right;

}

// If largest is not root

if (largest != i) {

swap(arr[i], arr[largest]);

// Recursively heapify the affected sub-tree

maxHeapify(arr, n, largest);

}

}

// Function to build a max heap

void buildMaxHeap(int arr[], int n) {

// Build heap (rearrange array)

for (int i = n / 2 - 1; i >= 0; i--)

{

maxHeapify(arr, n, i);

}

}

// Function to heapify a subtree rooted with node i which is an index in arr[]

void minHeapify(int arr[], int n, int i) {

int smallest = i; // Initialize smallest as root

int left = 2 \* i + 1; // left child

int right = 2 \* i + 2; // right child

// If left child is smaller than root

if (left < n && arr[left] < arr[smallest])

smallest = left;

// If right child is smaller than smallest so far

if (right < n && arr[right] < arr[smallest])

smallest = right;

// If smallest is not root

if (smallest != i) {

swap(arr[i], arr[smallest]);

// Recursively heapify the affected sub-tree

minHeapify(arr, n, smallest);

}

}

void buildMinHeap(int arr[], int n) {

// Rearrange Elements In array.

for (int i = n / 2 - 1; i >= 0; i--)

minHeapify(arr, n, i);

}

// print an array

void printArray(int arr[], int n) {

for (int i = 0; i < n; ++i)

cout << arr[i] << " ";

cout << "\n";

}

int main() {

int arr[] = { 8, 12, 9, 18, 5, 3, 9, 20, 10, 6 };

int n = sizeof(arr) / sizeof(arr[0]);

cout << "Original Array: ";

printArray(arr, n);

// Function Calling and Convert array to max heap

buildMaxHeap(arr, n);

cout << "Max Heap: ";

printArray(arr, n);

// Function Calling and Convert max heap to min heap

buildMinHeap(arr, n);

cout << "Min Heap: ";

printArray(arr, n);

system("pause");

return 0;

}

Output:

