Assignment #4

21F-9513

21I-0759

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:

Task #2:

#include <iostream>

using namespace std;

class TreeNode

{

public:

int val;

TreeNode\* left;

TreeNode\* right;

TreeNode(int v)

{

val=v;

left = nullptr;

right = nullptr;

}

};

class BST

{

public:

TreeNode\* sortingfunc(int sortedArray[], int first, int last)

{

if (first > last)

return nullptr;

int mid = (first + last) / 2;

TreeNode\* root = new TreeNode(sortedArray[mid]);

root->left = sortingfunc(sortedArray, first, mid - 1);

root->right = sortingfunc(sortedArray, mid + 1, last);

return root;

}

};

void inorder(TreeNode\* root)

{

if (root == nullptr)

return;

inorder(root->left);

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

inorder(root->right);

}

int main()

{

int sortedArray[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };

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

BST obj;

TreeNode\* root = obj.sortingfunc(sortedArray, 0, n - 1);

cout << "Inorder traversal of the minimal height BST= ";

inorder(root);

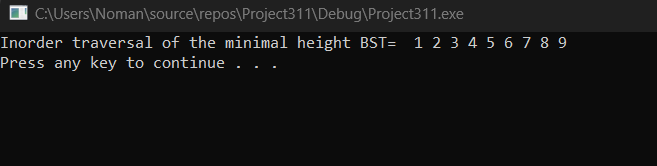
cout << endl;

system("pause");

return 0;

}

Output:



Task #3:

#include <iostream>

#include<algorithm>

using namespace std;

class AVLTree

{

private:

struct Node

{

int data;

Node\* left;

Node\* right;

int height;

Node(int val)

{

data = val;

left = nullptr;

right=nullptr;

height = 1;

}

};

Node\* root;

int height(Node\* node)

{

if (node == nullptr)

return 0;

return (node->height);

}

int getBalance(Node\* node)

{

if (node == nullptr)

return 0;

return height(node->left) - height(node->right);

}

Node\* rotateRight(Node\* y)

{

Node\* x = y->left;

Node\* T2 = x->right;

x->right = y;

y->left = T2;

y->height = 1 + max(height(y->left), height(y->right));

x->height = 1 + max(height(x->left), height(x->right));

return x;

}

Node\* rotateLeft(Node\* x)

{

Node\* y = x->right;

Node\* T2 = y->left;

y->left = x;

x->right = T2;

x->height = 1 + max(height(x->left), height(x->right));

y->height = 1 + max(height(y->left), height(y->right));

return y;

}

Node\* insert(Node\* node, int key)

{

if (node == nullptr)

return new Node(key);

if (key < node->data)

node->left = insert(node->left, key);

else if (key > node->data)

node->right = insert(node->right, key);

else

return node;

node->height = 1 + max(height(node->left), height(node->right));

int balance = getBalance(node);

if (balance > 1 && key < node->left->data)

return rotateRight(node);

if (balance < -1 && key > node->right->data)

return rotateLeft(node);

if (balance > 1 && key > node->left->data) {

node->left = rotateLeft(node->left);

return rotateRight(node);

}

if (balance < -1 && key < node->right->data) {

node->right = rotateRight(node->right);

return rotateLeft(node);

}

return node;

}

bool findingpair(Node\* root, int sum)

{

if (!root)

return false;

int currVal = root->data;

int remaining = sum - currVal;

if (remaining > currVal) //right subtree

{

if (findingpair(root->right, sum))

return true;

}

else if (remaining < currVal) //left subtree

{

if (findingpair(root->left, sum))

return true;

}

else

{

// found a pair with the given sum

return true;

}

return findingpair(root->left, sum) || findingpair(root->right, sum);

}

public:

AVLTree()

{

root = nullptr;

}

void insert(int key)

{

root = insert(root, key);

}

bool findPairWithSum(int sum)

{

return findingpair(root, sum);

}

};

int main()

{

AVLTree obj;

obj.insert(1);

obj.insert(3);

obj.insert(7);

obj.insert(9);

obj.insert(4);

obj.insert(6);

obj.insert(13);

int sum = 12;

if (obj.findPairWithSum(sum))

{

cout << "Found a pair with sum " << sum << " in the AVL tree" << endl;

}

else

{

cout << "No pair found with sum " << sum << " in the AVL tree" << endl;

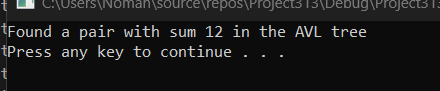
}

system("pause");

return 0;

}

Output:



Question#2:

Task#1:

#include<iostream>

#include<string>

using namespace std;

struct Stack

{

char data;

Stack\* next;

};

class InfixToPostfixConverter

{

public:

//FUNCTION TO CONVERT INFIX EXPRESSION TO POSTFIX EXPRESSION:-

string convertToPostfix(string infix)

{

string postfix;

Stack\* top = NULL;

// Push a left parenthesis to the stack

push(top, '(');

// Append a right parenthesis to the end of infix

infix += ')';

// Iterate through the characters in the infix expression

for (int i = 0; i < infix.length(); i++)

{

// Current character

char c = infix[i];

// If the character is a digit, add it to the postfix expression

if (isdigit(c))

postfix += c;

// If the character is a left parenthesis, push it to the stack

else if (c == '(')

push(top, c);

// If the character is an operator

else if (c == '+' || c == '-' || c == '\*' || c == '/' || c == '^')

{

// Pop operators from the stack while they have equal or higher precedence than the current operator

while (precedence(stackTop(top), c))

postfix += pop(top);

// Push the current character to the stack

push(top, c);

}

// If the character is a right parenthesis

else if (c == ')')

{

// Pop operators from the stack and add them to the postfix expression

while (stackTop(top) != '(')

postfix += pop(top);

// Pop and discard the left parenthesis from the stack

pop(top);

}

}

return postfix;

}

//FUNCTION TO DETERMINE PRECEDENCE OF OPERATORS:-

bool precedence(char operator1, char operator2)

{

// If operator1 has higher or equal precedence than operator2, return true

if ((operator1 == '+' || operator1 == '-') && (operator2 == '+' || operator2 == '-'))

return true;

else if ((operator1 == '\*' || operator1 == '/') && (operator2 == '+' || operator2 == '-' || operator2 == '\*'

|| operator2 == '/'))

return true;

else if (operator1 == '^' && (operator2 == '+' || operator2 == '-' || operator2 == '\*' || operator2 == '/' ||

operator2 == '^'))

return true;

else

return false;

}

//FUNCTION TO PUSH A VALUE TO THE STACK:-

void push(Stack\*& top, char data)

{

Stack\* newNode = new Stack;

newNode->data = data;

newNode->next = top;

top = newNode; //SETS THE NEW NODE AS THE NEW TOP.

}

//FUNCTION TO POP A VALUE FROM THE STACK:-

char pop(Stack\*& top)

{

// Store the top value

char data = top->data;

// Create a temporary node

Stack\* temp = top;

// Set the top to the next node

top = top->next;

// Delete the temp node

delete temp;

// Return the stored value

return data;

}

//FUNCTION TO RETURN THE TOP VALUE OF THE STACK WITHOUT POPPING:-

char stackTop(Stack\* top)

{

return top->data;

}

//FUNCTION TO CHECK IF THE STACK IS EMPTY:-

bool isEmpty(Stack\* top)

{

return (top == NULL);

}

};

int main()

{

InfixToPostfixConverter obj;

string input; //input will be infix expression.

string output; //output will be postfix expression.

cout << "Input Infix expression: ";

getline(cin, input);

output = obj.convertToPostfix(input);

cout << endl;

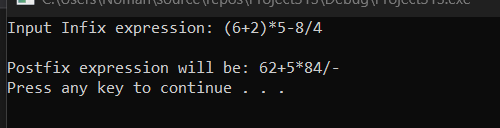
cout << "Postfix expression will be: " << output << endl;

system("pause");

return 0;

}

Output:



Task #2:

#include <iostream>

#include<string>

using namespace std;

class Stack

{

private:

char\* arr;

int top;

int capacity;

public:

Stack(int size) //constructor

{

top = -1;

capacity = size;

arr = new char[capacity];

}

void push(char ch)

{

if (top == capacity - 1)

{

cout << "Stack overflow!" << endl;

return;

}

arr[++top] = ch;

}

void pop()

{

if (top == -1)

{

cout << "Stack underflow!" << endl;

return;

}

top--;

}

bool isEmpty() {

return top == -1;

}

char getTop() {

if (top == -1) {

return '\0';

}

return arr[top];

}

};

bool areBracketsBalanced(string str)

{

Stack st(str.length());

for (char ch : str) {

if (ch == '(' || ch == '[' || ch == '{')

{

st.push(ch);

}

else if (ch == ')' || ch == ']' || ch == '}') {

char topChar = st.getTop();

if (st.isEmpty() || (ch == ')' && topChar != '(') || (ch == ']' && topChar != '[') || (ch == '}' && topChar != '{')) {

return false;

}

st.pop();

}

}

return st.isEmpty();

}

int main() {

string input;

cout << "Enter a string with brackets: ";

getline(cin, input);

if (areBracketsBalanced(input))

{

cout << "The brackets are balanced.\n";

}

else {

cout << "The brackets are not balanced.\n";

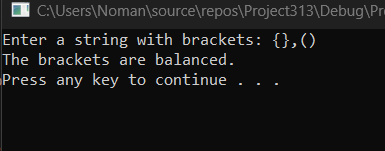
}

system("pause");

return 0;

}

Output:



Task #3:

#include <iostream>

#include<string>

using namespace std;

bool isPalindrome(string str)

{

int left = 0;

int right = str.length() - 1;

while (left < right) {

if (str[left] != str[right]) {

return false;

}

left++;

right--;

}

return true;

}

int main()

{

string input;

cout << "Enter a string: ";

getline(cin, input);

if (isPalindrome(input))

{

cout << "The string is a palindrome." << endl;

}

else {

cout << "The string is not a palindrome." << endl;

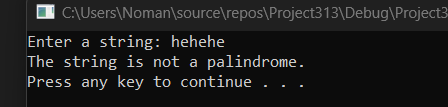
}

system("pause");

return 0;

}

Output:



Task #4: