**GROUP B**

**Construct an expression tree from the given prefix expression eg. +--a\*bc/def andtraverse it using postorder traversal(non recursive) and then delete the entire tree.**

#include <iostream>

#include <stack>

using namespace std;

// Node structure for expression tree

struct Node {

string data;

Node\* left;

Node\* right;

Node(string value) : data(value), left(nullptr), right(nullptr) {}

};

// Function to check if a character is an operand

bool isOperand(char c) {

return (c >= 'a' && c <= 'z') || (c >= 'A' && c <= 'Z');

}

// Function to construct an expression tree from prefix expression

Node\* constructExpressionTree(string prefixExpression) {

stack<Node\*> st;

int length = prefixExpression.length();

// Traverse the prefix expression in reverse order

for (int i = length - 1; i >= 0; i--) {

char ch = prefixExpression[i];

// If the character is an operand, create a new node and push it to the stack

if (isOperand(ch)) {

string operand(1, ch);

st.push(new Node(operand));

} else {

// If the character is an operator, pop two nodes from the stack,

// create a new node with the operator as the data, and push it back to the stack

Node\* operand1 = st.top();

st.pop();

Node\* operand2 = st.top();

st.pop();

string op(1, ch);

string expression = operand1->data + operand2->data + op;

Node\* operatorNode = new Node(expression);

operatorNode->left = operand1;

operatorNode->right = operand2;

st.push(operatorNode);

}

}

// The top of the stack will be the root of the expression tree

return st.top();

}

// Function to perform postorder traversal (non-recursive) on the expression tree

void postorderTraversal(Node\* root) {

if (root == nullptr)

return;

stack<Node\*> st;

Node\* current = root;

Node\* lastVisited = nullptr;

while (current || !st.empty()) {

if (current) {

st.push(current);

current = current->left;

} else {

Node\* topNode = st.top();

if (topNode->right && topNode->right != lastVisited) {

current = topNode->right;

} else {

cout << topNode->data << " ";

lastVisited = topNode;

st.pop();

}

}

}

}

// Function to delete the entire expression tree

void deleteExpressionTree(Node\* root) {

if (root == nullptr)

return;

deleteExpressionTree(root->left);

deleteExpressionTree(root->right);

delete root;

}

int main() {

string prefixExpression;

cout << "Enter the prefix expression: ";

cin >> prefixExpression;

Node\* root = constructExpressionTree(prefixExpression);

cout << "Postorder Traversal: ";

postorderTraversal(root);

cout << endl;

deleteExpressionTree(root);

return 0;

}

OUTPUT:-

Enter the prefix expression: +-\*abcdefghi+jklmnopqrstu/vwxyz

Postorder Traversal: a b ab\* c ab\*c- d ab\*c-d+