**1.a) The following is an infix expression. Change the infix expression into a prefix expression and evaluate the result of your output prefix expression by finding its value.**

**Sample Expression: (34\*2)- [{40/ (4-2)}+2]**

**Ans**

**Code:**

#include <iostream>

#include <stack>

#include <string>

#include <cmath>

#include <algorithm>

using namespace std;

// Function to check precedence of operators

int precedence(char o)

{

if (o == '+' || o == '-')

return 1;

if (o == '\*' || o == '/')

return 2;

if (o == '^')

return 3;

return 0;

}

// Function to check if a character is an operator

bool isOperators(char c)

{

return (c == '+' || c == '-' || c == '\*' || c == '/' || c == '^');

}

// Function to convert infix to prefix

string infixToPrefix(string infix)

{

stack<char> operators;

stack<string> operands;

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

{

char c = infix[i];

// If the character is an operand (number), push it to operands stack

if (isdigit(c))

{

string operand = "";

while (i < infix.size() && isdigit(infix[i]))

{

operand += infix[i];

i++;

}

i--; // Adjust index after exiting the loop

operands.push(operand);

}

// If the character is an opening bracket, push it to operators stack

else if (c == '(' || c == '[' || c == '{')

{

operators.push(c);

}

// If the character is a closing bracket

else if (c == ')' || c == ']' || c == '}')

{

char matchingBracket = (c == ')') ? '(' : (c == ']') ? '['

: '{';

while (!operators.empty() && operators.top() != matchingBracket)

{

char op = operators.top();

operators.pop();

string op2 = operands.top();

operands.pop();

string op1 = operands.top();

operands.pop();

string expr = op + op1 + op2;

operands.push(expr);

}

operators.pop(); // Pop the opening bracket

}

// If the character is an operator

else if (isOperators(c))

{

while (!operators.empty() && precedence(operators.top()) >= precedence(c))

{

char op = operators.top();

operators.pop();

string op2 = operands.top();

operands.pop();

string op1 = operands.top();

operands.pop();

string expr = op + op1 + op2;

operands.push(expr);

}

operators.push(c);

}

}

// Process remaining operators in the stack

while (!operators.empty())

{

char op = operators.top();

operators.pop();

string op2 = operands.top();

operands.pop();

string op1 = operands.top();

operands.pop();

string expr = op + op1 + op2;

operands.push(expr);

}

return operands.top();

}

// Function to evaluate a prefix expression

int evaluatePrefix(string prefix)

{

stack<int> samia;

// Traverse the prefix expression from right to left

for (int i = prefix.size() - 1; i >= 0; i--)

{

char c = prefix[i];

// If the character is an operand, push it to the stack

if (isdigit(c))

{

string operand = "";

while (i >= 0 && isdigit(prefix[i]))

{

operand = prefix[i] + operand;

i--;

}

i++; // Adjust index after exiting the loop

samia.push(stoi(operand));

}

// If the character is an operator, pop two elements, apply the operator, and push the result

else if (isOperators(c))

{

int op1 = samia.top();

samia.pop();

int op2 = samia.top();

samia.pop();

switch (c)

{

case '+':

samia.push(op1 + op2);

break;

case '-':

samia.push(op1 - op2);

break;

case '\*':

samia.push(op1 \* op2);

break;

case '/':

samia.push(op1 / op2);

break;

case '^':

samia.push(pow(op1, op2));

break;

}

}

}

return samia.top();

}

int main()

{

string infix = "(34\*2)-[{40/(4-2)}+2]";

// Convert infix to prefix

string prefix = infixToPrefix(infix);

cout << "Prefix Expression: " << prefix << endl;

// Evaluate the prefix expression

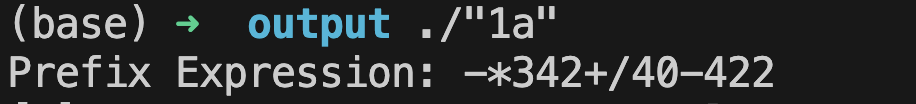
int result = evaluatePrefix(prefix);

cout << "Result: " << result << endl;

return 0;

}

**Output:**



**1.b) Create a queue using linked list and reverse its elements. Print the elements in two directions:**

**(i) Front to rear (ii) Rear to front**

**Ans**

**Code:**

#include <iostream>

#include <stack>

using namespace std;

// Node structure

struct N

{

int d;

N \*next;

N(int v) : d(v), next(nullptr) {}

};

// Queue class using Linked List

class Q

{

private:

N \*front, \*rear;

public:

Q() : front(nullptr), rear(nullptr) {}

// Enqueue operation

void enq(int v)

{

N \*newN = new N(v);

if (!rear)

{

front = rear = newN;

}

else

{

rear->next = newN;

rear = newN;

}

}

// Dequeue operation

int deq()

{

if (!front)

{

cout << "Queue is empty!" << endl;

return -1;

}

int v = front->d;

N \*temp = front;

front = front->next;

if (!front)

rear = nullptr;

delete temp;

return v;

}

// Print elements from front to rear

void printFrontToRear()

{

N \*temp = front;

cout << "Queue (Front to Rear): ";

while (temp)

{

cout << temp->d << " ";

temp = temp->next;

}

cout << endl;

}

// Reverse the queue

void reverse()

{

stack<int> samia;

while (front)

{

samia.push(deq());

}

while (!samia.empty())

{

enq(samia.top());

samia.pop();

}

}

// Print elements from rear to front

void printRearToFront()

{

reverse(); // Reverse the queue

printFrontToRear(); // Print in reverse order

reverse(); // Restore original order

}

};

int main()

{

Q queue;

int n, val;

// Input number of elements

cout << "Enter the number of elements to enqueue: ";

cin >> n;

// Input elements

cout << "Enter the elements:\n";

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

{

cin >> val;

queue.enq(val);

}

// Print Front to Rear

cout << "\nQueue printed from Front to Rear:\n";

queue.printFrontToRear();

// Print Rear to Front

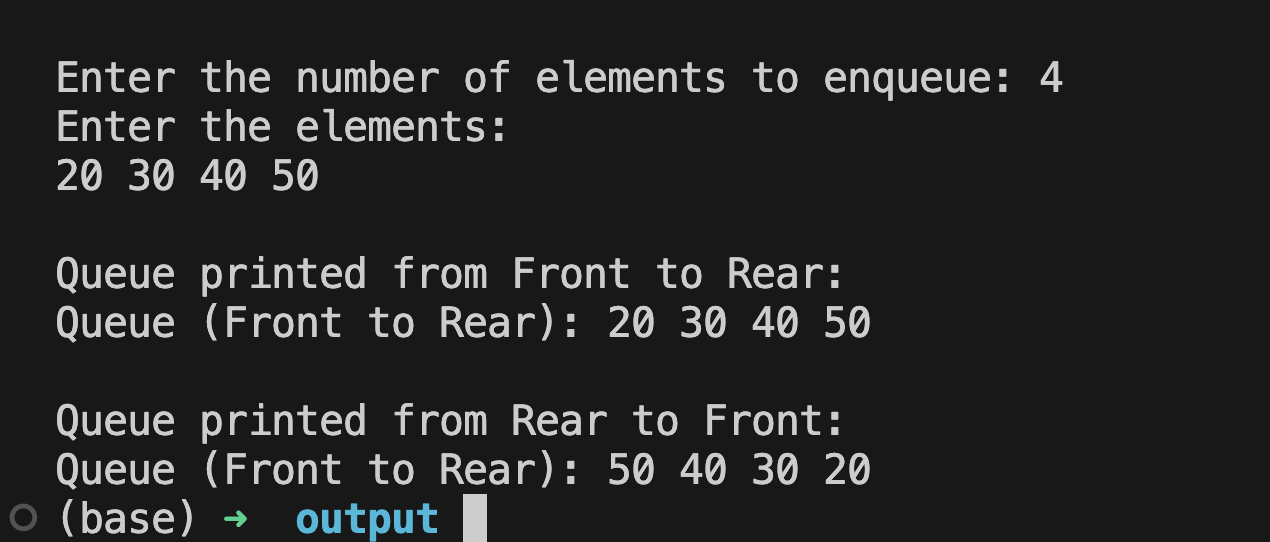
cout << "\nQueue printed from Rear to Front:\n";

queue.printRearToFront();

return 0;

}

**Output:**



**2.a) Change an infix expression to a postfix expression with the help of stack using linked list. Sample input: ((A+B)-C\*(D/E)) +F**

**Sample Output: AB+CDE/\*-F+**

**Ans**

**Code:**

#include <iostream>

#include <string>

using namespace std;

// Node structure for linked list

struct N

{

char d;

N \*next;

N(char val) : d(val), next(nullptr) {}

};

// Stack class implemented using linked list

class Stack

{

private:

N \*top;

public:

Stack() : top(nullptr) {}

// Push operation

void push(char val)

{

N \*newN = new N(val);

newN->next = top;

top = newN;

}

// Pop operation

char pop()

{

if (isEmpty())

{

cout << "Stack is empty!" << endl;

return '\0';

}

char val = top->d;

N \*temp = top;

top = top->next;

delete temp;

return val;

}

// Peek operation

char peek()

{

if (isEmpty())

return '\0';

return top->d;

}

// Check if stack is empty

bool isEmpty()

{

return top == nullptr;

}

};

// Function to check precedence of operators

int precedence(char op)

{

if (op == '+' || op == '-')

return 1;

if (op == '\*' || op == '/')

return 2;

if (op == '^')

return 3;

return 0;

}

// Function to check if a character is an operator

bool isOperator(char c)

{

return (c == '+' || c == '-' || c == '\*' || c == '/' || c == '^');

}

// Function to check if a character is an operand

bool isOperand(char c)

{

return (isalpha(c) || isdigit(c));

}

// Function to convert infix to postfix

string infixToPostfix(string infix)

{

Stack stack;

string postfix = "";

for (char c : infix)

{

// If the character is an operand, add it to the postfix string

if (isOperand(c))

{

postfix += c;

}

// If the character is an opening parenthesis, push it onto the stack

else if (c == '(')

{

stack.push(c);

}

// If the character is a closing parenthesis, pop and add to postfix until '(' is found

else if (c == ')')

{

while (!stack.isEmpty() && stack.peek() != '(')

{

postfix += stack.pop();

}

stack.pop(); // Remove '('

}

// If the character is an operator

else if (isOperator(c))

{

while (!stack.isEmpty() && precedence(stack.peek()) >= precedence(c))

{

postfix += stack.pop();

}

stack.push(c);

}

}

// Pop all remaining operators in the stack

while (!stack.isEmpty())

{

postfix += stack.pop();

}

return postfix;

}

int main()

{

string infix;

// Input infix expression from user

cout << "Enter an infix expression: ";

cin >> infix;

// Convert infix to postfix

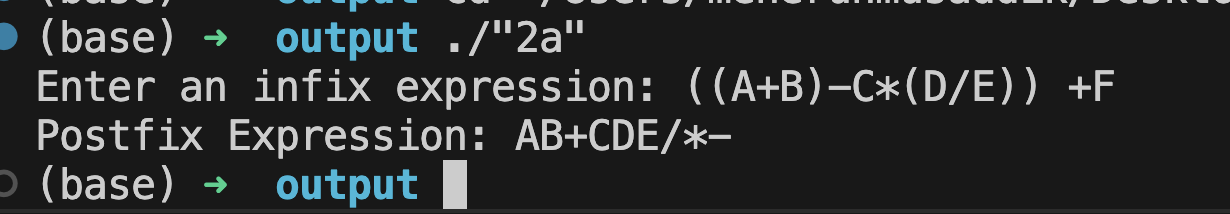
string postfix = infixToPostfix(infix);

// Output the postfix expression

cout << "Postfix Expression: " << postfix << endl;

return 0;

}  
  
**Output:**



**2.b) Create a binary search tree using an iterative approach for insertion function and deletion function. Print out the nodes of your constructed BST using any traversal mechanism. Now determine the height of the BST of the root node.**

**Formula to find height: |Left subtree| – |Right subtree| or |Right subtree| –|Left subtree|**

**Ans**

**Code:**

#include <iostream>

#include <cmath>

using namespace std;

// Node structure

struct N

{

int key;

N \*left;

N \*right;

N(int value) : key(value), left(nullptr), right(nullptr) {}

};

// Binary Search Tree class

class BST

{

private:

N \*root;

// Helper for in-order traversal

void inorderTraversal(N \*node)

{

if (!node)

return;

inorderTraversal(node->left);

cout << node->key << " ";

inorderTraversal(node->right);

}

// Helper to calculate height of the tree

int calculateHeight(N \*node)

{

if (!node)

return 0;

int leftHeight = calculateHeight(node->left);

int rightHeight = calculateHeight(node->right);

return max(leftHeight, rightHeight) + 1;

}

public:

BST() : root(nullptr) {}

// Iterative Insertion

void insert(int key)

{

N \*newN = new N(key);

if (!root)

{

root = newN;

return;

}

N \*current = root;

N \*parent = nullptr;

while (current)

{

parent = current;

if (key < current->key)

current = current->left;

else

current = current->right;

}

if (key < parent->key)

parent->left = newN;

else

parent->right = newN;

}

// Iterative Deletion

void deleteNode(int key)

{

N \*current = root;

N \*parent = nullptr;

// Find the node to delete

while (current && current->key != key)

{

parent = current;

if (key < current->key)

current = current->left;

else

current = current->right;

}

if (!current)

return; // Key not found

// Case 1: Node with only one child or no child

if (!current->left || !current->right)

{

N \*newChild = current->left ? current->left : current->right;

if (!parent) // Deleting root node

root = newChild;

else if (current == parent->left)

parent->left = newChild;

else

parent->right = newChild;

delete current;

}

// Case 2: Node with two children

else

{

N \*successorParent = current;

N \*successor = current->right;

// Find the inorder successor (smallest in the right subtree)

while (successor->left)

{

successorParent = successor;

successor = successor->left;

}

// Replace current's key with successor's key

current->key = successor->key;

// Delete the successor

if (successorParent->left == successor)

successorParent->left = successor->right;

else

successorParent->right = successor->right;

delete successor;

}

}

// In-order Traversal

void inorder()

{

inorderTraversal(root);

cout << endl;

}

// Calculate Height

int getHeight()

{

return calculateHeight(root);

}

// Calculate Height Difference

int getHeightDifference()

{

if (!root)

return 0;

int leftHeight = calculateHeight(root->left);

int rightHeight = calculateHeight(root->right);

return abs(leftHeight - rightHeight);

}

};

// Main Function

int main()

{

BST bst;

// Insert values

int values[] = {50, 30, 70, 20, 40, 60, 80};

for (int value : values)

bst.insert(value);

cout << "In-order Traversal of BST: ";

bst.inorder();

// Delete a value

bst.deleteNode(70);

cout << "In-order Traversal after deletion: ";

bst.inorder();

// Height of BST

cout << "Height of BST: " << bst.getHeight() << endl;

// Height Difference

cout << "Height Difference of root: " << bst.getHeightDifference() << endl;

return 0;

}

**Output:**  
