**1.Stack push, pop,peek implementation using array**

#include <iostream>

using namespace std;

// Define constants

const int MAX\_SIZE = 5;

int stack[MAX\_SIZE];

int top = -1;

// Push function to add an element to the stack

void push(int value) {

if (top == MAX\_SIZE - 1) {

cout << "Stack Overflow: Cannot push " << value << endl;

} else {

top++;

stack[top] = value;

cout << value << " pushed into the stack." << endl;

}

}

// Pop function to remove an element from the stack

int pop() {

if (top == -1) {

cout << "Stack Underflow: Stack is empty." << endl;

return -1; // Return an error value

} else {

int poppedValue = stack[top];

top--;

cout << poppedValue << " popped from the stack." << endl;

return poppedValue;

}

}

// Peek function to view the top element of the stack

int peek() {

if (top == -1) {

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

return -1; // Return an error value

} else {

return stack[top];

}

}

// Check if the stack is empty

bool isEmpty() {

return top == -1;

}

int main() {

// Push elements into the stack

push(10);

push(20);

push(30);

push(40);

push(50);

push(60); // This will cause Stack Overflow

cout << "Top element is: " << peek() << endl;

// Pop elements from the stack

pop();

pop();

pop();

pop();

pop();

pop(); // This will cause Stack Underflow

return 0;

}

**2.Stack implementation user input push,pop,peek**

#include <iostream>

using namespace std;

// Define constants

const int MAX\_SIZE = 5;

int stack[MAX\_SIZE];

int top = -1;

// Push function to add an element to the stack

void push(int value) {

if (top == MAX\_SIZE - 1) {

cout << "Stack Overflow: Cannot push " << value << endl;

} else {

top++;

stack[top] = value;

cout << value << " pushed into the stack." << endl;

}

}

// Pop function to remove an element from the stack

int pop() {

if (top == -1) {

cout << "Stack Underflow: Stack is empty." << endl;

return -1; // Return an error value

} else {

int poppedValue = stack[top];

top--;

cout << poppedValue << " popped from the stack." << endl;

return poppedValue;

}

}

// Peek function to view the top element of the stack

int peek() {

if (top == -1) {

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

return -1; // Return an error value

} else {

return stack[top];

}

}

// Check if the stack is empty

bool isEmpty() {

return top == -1;

}

int main() {

int choice, value;

do {

cout << "\nStack Operations Menu:\n";

cout << "1. Push\n";

cout << "2. Pop\n";

cout << "3. Peek\n";

cout << "4. Check if Empty\n";

cout << "5. Exit\n";

cout << "Enter your choice: ";

cin >> choice;

switch (choice) {

case 1:

cout << "Enter the value to push: ";

cin >> value;

push(value);

break;

case 2:

pop();

break;

case 3:

cout << "Top element is: " << peek() << endl;

break;

case 4:

if (isEmpty()) {

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

} else {

cout << "Stack is not empty." << endl;

}

break;

case 5:

cout << "Exiting program." << endl;

break;

default:

cout << "Invalid choice. Please try again." << endl;

}

} while (choice != 5);

return 0;

}

**3.Stack implementation using liked list push,pop,peek**

#include <iostream>

#include<cstdlib>

#include<cctype>

using namespace std;

// Define a Node structure for the linked list

struct Node {

int data;

Node\* next;

};

// Push function to add an element to the stack

void push(Node\*& top, int value) {

Node\* newNode;

newNode=(Node\*)malloc(sizeof(Node)); // Create a new node

newNode->data = value;

newNode->next = top; // Link the new node to the current top

top = newNode; // Update the top to the new node

cout << value << " pushed into the stack." << endl;

}

// Pop function to remove the top element from the stack

int pop(Node\*& top) {

if (top == nullptr) {

cout << "Stack Underflow: Stack is empty." << endl;

return -1; // Return an error value

} else {

Node\* temp = top; // Save the current top node

int poppedValue = top->data;

top = top->next; // Update the top to the next node

delete temp; // Free the memory of the removed node

cout << poppedValue << " popped from the stack." << endl;

return poppedValue;

}

}

// Peek function to view the top element of the stack

int peek(Node\* top) {

if (top == nullptr) {

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

return -1; // Return an error value

} else {

return top->data;

}

}

// Check if the stack is empty

bool isEmpty(Node\* top) {

return top == nullptr;

}

int main() {

Node\* top = nullptr; // Initialize the stack as empty

// Push elements into the stack

push(top, 10);

push(top, 20);

push(top, 30);

// Display the top element

cout << "Top element is: " << peek(top) << endl;

// Pop elements from the stack

pop(top);

pop(top);

// Display the top element after popping

cout << "Top element is: " << peek(top) << endl;

// Check if the stack is empty

if (isEmpty(top)) {

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

} else {

cout << "Stack is not empty." << endl;

}

// Pop the last element

pop(top);

// Try to pop from an empty stack

pop(top);

return 0;

}

**4. Queue implementation insert and delete function also overflow and underflow using array**

#include <iostream>

using namespace std;

const int MAX\_SIZE = 5; // Maximum size of the queue

int queue[MAX\_SIZE];

int front = -1, rear = -1;

// Function to insert an element into the queue

void insert(int value) {

if (rear == MAX\_SIZE - 1) {

cout << "Queue Overflow: Cannot insert " << value << endl;

}

else {

if (front == -1) front = 0; // Initialize front on the first insertion

rear++;

queue[rear] = value;

cout << value << " inserted into the queue." << endl;

}

}

// Function to delete an element from the queue

int deleteElement() {

if (front == -1 || front > rear) {

cout << "Queue Underflow: Queue is empty." << endl;

return -1; // Return an error value

} else {

int deletedValue = queue[front];

cout << deletedValue << " deleted from the queue." << endl;

front++;

if (front > rear) { // Reset queue if it becomes empty

front = -1;

rear = -1;

}

return deletedValue;

}

}

// Function to display the elements of the queue

void display() {

if (front == -1 || front > rear) {

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

} else {

cout << "Queue elements: ";

for (int i = front; i <= rear; i++) {

cout << queue[i] << " ";

}

cout << endl;

}

}

int main() {

insert(10);

insert(20);

insert(30);

insert(40);

insert(50);

insert(60); // This will cause Queue Overflow

display();

deleteElement();

deleteElement();

display();

deleteElement();

deleteElement();

deleteElement();

deleteElement(); // This will cause Queue Underflow

return 0;

}

**5.Queue implementation insert and delete function also overflow and underflow using linked list**

#include <iostream>

#include<cstdlib>

#include<cctype>

using namespace std;

// Node structure for the linked list

struct Node {

int data;

Node\* next;

};

// Function to insert an element into the queue

void insert(Node\*& front, Node\*& rear, int value) {

Node\* newNode = new Node(); // Create a new node

newNode->data = value;

newNode->next = nullptr;

if (rear == nullptr) { // If the queue is empty

front = rear = newNode;

} else {

rear->next = newNode; // Link the new node at the end

rear = newNode; // Update the rear pointer

}

cout << value << " inserted into the queue." << endl;

}

// Function to delete an element from the queue

int deleteElement(Node\*& front, Node\*& rear) {

if (front == nullptr) { // If the queue is empty

cout << "Queue Underflow: Queue is empty." << endl;

return -1; // Return an error value

}

Node\* temp = front; // Save the current front node

int deletedValue = front->data;

front = front->next; // Update the front pointer

if (front == nullptr) { // If the queue becomes empty

rear = nullptr;

}

delete temp; // Free memory of the removed node

cout << deletedValue << " deleted from the queue." << endl;

return deletedValue;

}

// Function to display elements in the queue

void display(Node\* front) {

if (front == nullptr) {

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

} else {

cout << "Queue elements: ";

Node\* temp = front;

while (temp != nullptr) {

cout << temp->data << " ";

temp = temp->next;

}

cout << endl;

}

}

int main() {

Node\* front = nullptr; // Initialize front and rear pointers

Node\* rear = nullptr;

// Insert elements into the queue

insert(front, rear, 10);

insert(front, rear, 20);

insert(front, rear, 30);

// Display elements in the queue

display(front);

// Delete elements from the queue

deleteElement(front, rear);

deleteElement(front, rear);

// Display elements after deletion

display(front);

deleteElement(front, rear);

deleteElement(front, rear); // This will cause Queue Underflow

return 0;

}

**6. Reverse a string using stack**

#include <iostream>

#include<cstdlib>

#include<cctype>

using namespace std;

// A structure to represent a stack

struct Stack {

int top;

char array[5]; // Fixed size array of 5

};

// Function to create a stack

Stack\* createStack() {

Stack\* stack = new Stack();

stack->top = -1;

return stack;

}

// Stack is full when top is equal to 4 (last index)

int isFull(Stack\* stack) {

return stack->top == 4;

}

// Stack is empty when top is equal to -1

int isEmpty(Stack\* stack) {

return stack->top == -1;

}

// Function to add an item to stack

void push(Stack\* stack, char item) {

if (isFull(stack))

return;

stack->array[++stack->top] = item;

}

// Function to remove an item from stack

char pop(Stack\* stack) {

if (isEmpty(stack))

return -1;

return stack->array[stack->top--];

}

// A stack-based function to reverse a string

void reverse(char str[]) {

// Create a stack with fixed size

Stack\* stack = createStack();

// Push all characters of string to stack

for (int i = 0; i < 5; i++) { // Loop limited to 5 due to fixed size

push(stack, str[i]);

}

// Pop all characters of string and put them back to str

for (int i = 0; i < 5; i++) {

str[i] = pop(stack);

}

}

// Driver code

int main() {

char str[] = "ABCDE"; // String size should not exceed 5 characters

reverse(str);

cout << "Reversed string is " << str;

return 0;

}

**7.Reverse an array using stack**

#include <iostream>

#include<cstdlib>

#include<cctype>

using namespace std;

// Structure of stack

struct Stack {

// Stores index of top element of a stack

int top;

// Stores maximum count of elements stored in a stack

unsigned capacity;

// Stores address of array element

int\* array;

};

// Function to initialize a stack of given capacity

Stack\* createStack(unsigned capacity) {

Stack\* stack = new Stack();

stack->capacity = capacity;

stack->top = -1;

stack->array = new int[capacity];

return stack;

}

// Function to check if the stack is full or not

int isFull(Stack\* stack) {

return stack->top == stack->capacity - 1;

}

// Function to check if the stack is empty or not

int isEmpty(Stack\* stack) {

return stack->top == -1;

}

// Function to insert an element into the stack

void push(Stack\* stack, int item) {

// If stack is full, do nothing

if (isFull(stack))

return;

// Insert element into stack

stack->array[++stack->top] = item;

}

// Function to remove an element from the stack

int pop(Stack\* stack) {

// If stack is empty, return -1

if (isEmpty(stack))

return -1;

// Pop element from stack

return stack->array[stack->top--];

}

// Function to reverse the array elements

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

// Initialize a stack of capacity n

Stack\* stack = createStack(n);

// Push all elements of the array into the stack

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

push(stack, arr[i]);

}

// Pop elements from the stack to reverse the array

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

arr[i] = pop(stack);

}

// Print the reversed array elements

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

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

}

}

// Driver Code

int main() {

int arr[] = { 100, 200, 300, 400 };

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

reverseArray(arr, N);

return 0;

}

**8. Balanced parenthesis**

#include <bits/stdc++.h>

using namespace std;

#define MAX\_SIZE 100

char s[MAX\_SIZE];

int top = -1;

void push(char item) {

if (top >= MAX\_SIZE - 1) {

cout << "Stack Overflow\n";

return;

}

s[++top] = item;

}

char pop() {

if (top < 0) {

cout << "Stack Underflow\n";

return '\0';

}

return s[top--];

}

bool stack\_is\_empty() {

return top == -1;

}

bool is\_matching\_pair(char open, char close) {

return (open == '(' && close == ')') ||

(open == '{' && close == '}') ||

(open == '[' && close == ']');

}

bool ispar(const string& s) {

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

if (s[i] == '(' || s[i] == '{' || s[i] == '[') {

push(s[i]);

} else if (s[i] == ')' || s[i] == '}' || s[i] == ']') {

if (!stack\_is\_empty() && is\_matching\_pair(pop(), s[i])) {

continue;

} else {

return false;

}

}

}

return stack\_is\_empty();

}

int main() {

string s;

cout << "Enter an expression: ";

cin >> s;

if (ispar(s))

cout << "Balanced\n";

else

cout << "Not Balanced\n";

return 0;

}

**9. Circular queue**

#include <bits/stdc++.h>

using namespace std;

#define MAX\_SIZE 100

int q[MAX\_SIZE];

int front = -1;

int rear = -1;

bool isFull() {

return (front == 0 && rear == MAX\_SIZE - 1) || (front == rear + 1);

}

bool isEmpty() {

return front == -1;

}

void enqueue(int item) {

if (isFull()) {

cout << "Queue Overflow\n";

return;

}

if (front == -1) {

front = 0;

rear = 0;

} else if (rear == MAX\_SIZE - 1) {

rear = 0;

} else {

rear++;

}

q[rear] = item;

}

int dequeue() {

if (isEmpty()) {

cout << "Queue Underflow\n";

return -1;

}

int item = q[front];

if (front == rear) {

front = -1;

rear = -1;

} else if (front == MAX\_SIZE - 1) {

front = 0;

} else {

front++;

}

return item;

}

void displayQueue() {

if (isEmpty()) {

cout << "Queue is empty\n";

return;

}

cout << "Queue elements: ";

if (rear >= front) {

for (int i = front; i <= rear; i++) {

cout << q[i] << " ";

}

} else {

for (int i = front; i < MAX\_SIZE; i++) {

cout << q[i] << " ";

}

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

cout << q[i] << " ";

}

}

cout << endl;

}

int main() {

int choice, item;

while (true) {

cout << "\nChoose an operation:\n";

cout << "1. Enqueue\n";

cout << "2. Dequeue\n";

cout << "3. Display Queue\n";

cout << "4. Exit\n";

cin >> choice;

switch (choice) {

case 1:

cout << "Enter value to enqueue: ";

cin >> item;

enqueue(item);

break;

case 2:

item = dequeue();

if (item != -1) {

cout << "Dequeued value: " << item << endl;

}

break;

case 3:

displayQueue();

break;

case 4:

return 0;

default:

cout << "Invalid choice. Please try again.\n";

}

}

}

**10. Queue using stack (Deque costly)**

#include <bits/stdc++.h>

using namespace std;

#define MAX\_SIZE 100

int stack1[MAX\_SIZE], stack2[MAX\_SIZE];

int top1 = -1, top2 = -1;

void push1(int data) {

if (top1 == MAX\_SIZE - 1) {

cout << "Stack Overflow" << endl;

return;

}

stack1[++top1] = data;

}

int pop1() {

if (top1 == -1) {

cout << "Stack Underflow" << endl;

return -1;

}

return stack1[top1--];

}

void push2(int data) {

if (top2 == MAX\_SIZE - 1) {

cout << "Stack Overflow" << endl;

return;

}

stack2[++top2] = data;

}

int pop2() {

if (top2 == -1) {

cout << "Stack Underflow" << endl;

return -1;

}

return stack2[top2--];

}

void enqueue(int data) {

push1(data);

}

void dequeue() {

if (top1 == -1 && top2 == -1) {

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

return;

}

while (top1 != -1) {

push2(pop1());

}

int dequeuedElement = pop2();

cout << "Dequeued: " << dequeuedElement << endl;

while (top2 != -1) {

push1(pop2());

}

}

void display() {

if (top1 == -1) {

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

return;

}

for (int i = 0; i <= top1; i++) {

cout << stack1[i] << " ";

}

cout << endl;

}

int main() {

enqueue(10);

enqueue(20);

enqueue(30);

dequeue();

enqueue(40);

enqueue(50);

cout << "Queue contents: ";

display();

return 0;

}

**11. Queue using stack(Enque costly)**

#include <bits/stdc++.h>

using namespace std;

#define MAX\_SIZE 100

int stack1[MAX\_SIZE], stack2[MAX\_SIZE];

int top1 = -1, top2 = -1;

void push1(int data) {

if (top1 == MAX\_SIZE - 1) {

cout << "Stack Overflow" << endl;

return;

}

stack1[++top1] = data;

}

int pop1() {

if (top1 == -1) {

cout << "Stack Underflow" << endl;

return -1;

}

return stack1[top1--];

}

void push2(int data) {

if (top2 == MAX\_SIZE - 1) {

cout << "Stack Overflow" << endl;

return;

}

stack2[++top2] = data;

}

int pop2() {

if (top2 == -1) {

cout << "Stack Underflow" << endl;

return -1;

}

return stack2[top2--];

}

void enqueue(int data) {

while (top1 != -1) {

push2(pop1());

}

push1(data);

while (top2 != -1) {

push1(pop2());

}

}

int dequeue() {

if (top1 == -1) {

cout << "Queue Underflow" << endl;

return -1;

}

return pop1();

}

void display() {

if (top1 == -1) {

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

return;

}

for (int i = top1; i >= 0; i--) {

cout << stack1[i] << " ";

}

cout << endl;

}

int main() {

enqueue(1);

enqueue(2);

enqueue(3);

cout << "Dequeued: " << dequeue() << endl;

enqueue(4);

enqueue(5);

cout << "Queue contents: ";

display();

return 0;

}

**12. Reverse individual word**

#include <bits/stdc++.h>

using namespace std;

#define MAX\_SIZE 100

char s[MAX\_SIZE];

int top = -1;

void push(char item) {

if (top >= MAX\_SIZE - 1) {

cout << "Stack Overflow\n";

return;

}

s[++top] = item;

}

char pop() {

if (top < 0) {

cout << "Stack Underflow\n";

return '\0';

}

return s[top--];

}

bool stack\_is\_empty() {

return top == -1;

}

void reverseWords(string& str) {

string reversed = "";

string word = "";

for (char c : str) {

if (c == ' ') {

while (!stack\_is\_empty()) {

word += pop();

}

reversed += word + ' ';

word = "";

} else {

push(c);

}

}

while (!stack\_is\_empty()) {

word += pop();

}

reversed += word;

str = reversed;

}

int main() {

string str;

cout << "Enter a string to reverse: ";

getline(cin, str);

reverseWords(str);

cout << "Reversed string: " << str << endl;

return 0;

}

**13. Reverse string**

#include <bits/stdc++.h>

using namespace std;

#define MAX\_SIZE 100

char s[MAX\_SIZE];

int top = -1;

void push(char item) {

if (top >= MAX\_SIZE - 1) {

cout << "Stack Overflow\n";

return;

}

s[++top] = item;

}

char pop() {

if (top < 0) {

cout << "Stack Underflow\n";

return '\0';

}

return s[top--];

}

bool is\_empty() {

return top == -1;

}

void reverseString(string& str) {

for (char c : str) {

push(c);

}

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

str[i] = pop();

}

}

int main() {

string str;

cout << "Enter a string to reverse: ";

getline(cin, str);

reverseString(str);

cout << "Reversed string: " << str << endl;

return 0;

}

**14. Stack sorting**

#include <bits/stdc++.h>

using namespace std;

#define MAX\_SIZE 100

int inputStack[MAX\_SIZE];

int tmpStack[MAX\_SIZE];

int topInput = -1;

int topTmp = -1;

void pushInput(int item) {

if (topInput >= MAX\_SIZE - 1) {

cout << "Input Stack Overflow\n";

return;

}

inputStack[++topInput] = item;

}

int popInput() {

if (topInput < 0) {

cout << "Input Stack Underflow\n";

return -1;

}

return inputStack[topInput--];

}

void pushTmp(int item) {

if (topTmp >= MAX\_SIZE - 1) {

cout << "Temporary Stack Overflow\n";

return;

}

tmpStack[++topTmp] = item;

}

int popTmp() {

if (topTmp < 0) {

cout << "Temporary Stack Underflow\n";

return -1;

}

return tmpStack[topTmp--];

}

void sortStack() {

while (topInput != -1) {

int temp = popInput();

while (topTmp != -1 && tmpStack[topTmp] > temp) {

pushInput(popTmp());

}

pushTmp(temp);

}

while (topTmp != -1) {

pushInput(popTmp());

}

}

void displayStack() {

if (topInput == -1) {

cout << "Stack is empty\n";

return;

}

for (int i = topInput; i >= 0; i--) {

cout << inputStack[i] << " ";

}

cout << endl;

}

int main() {

pushInput(34);

pushInput(3);

pushInput(31);

pushInput(98);

pushInput(92);

pushInput(23);

cout << "Stack before sorting: ";

displayStack();

sortStack();

cout << "Stack after sorting: ";

displayStack();

return 0;

}

**15. Stack using queue (pop costly)**

#include <iostream>

using namespace std;

#define MAX\_SIZE 100

int q1[MAX\_SIZE], q2[MAX\_SIZE];

int front1 = 0, rear1 = -1, size1 = 0;

int front2 = 0, rear2 = -1, size2 = 0;

void enqueue(int x, int &rear, int &size, int queue[]) {

if (size == MAX\_SIZE) {

cout << "Queue Overflow" << endl;

return;

}

queue[++rear] = x;

size++;

}

int dequeue(int &front, int &size, int queue[]) {

if (size == 0) {

cout << "Queue Underflow" << endl;

return -1;

}

int item = queue[front++];

size--;

return item;

}

bool isEmpty(int size) {

return size == 0;

}

int peek(int front, int size, int queue[]) {

if (size == 0) {

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

return -1;

}

return queue[front];

}

void push(int x) {

enqueue(x, rear1, size1, q1);

}

int pop() {

if (isEmpty(size1)) {

cout << "Stack Underflow" << endl;

return -1;

}

int currentSize = size1;

for (int i = 0; i < currentSize - 1; i++) {

enqueue(dequeue(front1, size1, q1), rear2, size2, q2);

}

int item = dequeue(front1, size1, q1);

for (int i = 0; i < currentSize - 1; i++) {

enqueue(dequeue(front2, size2, q2), rear1, size1, q1);

}

return item;

}

int top() {

if (isEmpty(size1)) {

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

return -1;

}

int currentSize = size1;

for (int i = 0; i < currentSize - 1; i++) {

enqueue(dequeue(front1, size1, q1), rear2, size2, q2);

}

int item = peek(front1, size1, q1);

for (int i = 0; i < currentSize - 1; i++) {

enqueue(dequeue(front2, size2, q2), rear1, size1, q1);

}

return item;

}

bool isStackEmpty() {

return isEmpty(size1);

}

int main() {

push(10);

push(20);

push(30);

cout << "Top element: " << top() << endl;

cout << "Popped: " << pop() << endl;

cout << "Popped: " << pop() << endl;

push(40);

cout << "Top element: " << top() << endl;

cout << "Popped: " << pop() << endl;

cout << "Popped: " << pop() << endl;

if (isStackEmpty())

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

return 0;

}

**16. Stack using queue ( Push costly)**

#include <bits/stdc++.h>

using namespace std;

#define MAX\_SIZE 100

int q1[MAX\_SIZE], q2[MAX\_SIZE];

int front1 = 0, rear1 = -1, size1 = 0;

int front2 = 0, rear2 = -1, size2 = 0;

void enqueue(int queue[], int &rear, int &size, int x) {

if (size == MAX\_SIZE) {

cout << "Queue Overflow" << endl;

return;

}

queue[++rear] = x;

size++;

}

int dequeue(int queue[], int &front, int &size) {

if (size == 0) {

cout << "Queue Underflow" << endl;

return -1;

}

int item = queue[front++];

size--;

return item;

}

bool isEmpty(int size) {

return size == 0;

}

int peek(int queue[], int front, int size) {

if (size == 0) {

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

return -1;

}

return queue[front];

}

// Push operation (costly O(n))

void push(int x) {

if (isEmpty(size1)) {

enqueue(q1, rear1, size1, x);

} else {

int currentSize = size1;

// Move all elements from Q1 to Q2

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

enqueue(q2, rear2, size2, dequeue(q1, front1, size1));

}

// Enqueue the new element into Q1

enqueue(q1, rear1, size1, x);

// Move all elements back from Q2 to Q1

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

enqueue(q1, rear1, size1, dequeue(q2, front2, size2));

}

}

}

// Pop operation (cheap O(1))

int pop() {

if (isEmpty(size1)) {

cout << "Stack Underflow" << endl;

return -1;

}

return dequeue(q1, front1, size1);

}

// Peek operation

int top() {

if (isEmpty(size1)) {

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

return -1;

}

return peek(q1, front1, size1);

}

// Check if stack is empty

bool isStackEmpty() {

return isEmpty(size1);

}

int main() {

push(10);

push(20);

push(30);

cout << "Top element: " << top() << endl; // 30

cout << "Popped: " << pop() << endl; // 30

cout << "Popped: " << pop() << endl; // 20

push(40);

cout << "Top element: " << top() << endl; // 40

cout << "Popped: " << pop() << endl; // 40

cout << "Popped: " << pop() << endl; // 10

if (isStackEmpty())

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

return 0;

}

**17. BFS algorithm**

#include <bits/stdc++.h>

using namespace std;

int totalNodes = 100;

void addEdge(vector<vector<int>>& graph, int u, int v) {

    graph[u].push\_back(v);

    graph[v].push\_back(u);

}

void performBFS(vector<vector<int>>& graph, int startNode) {

    vector<bool> visited(totalNodes, false);

    queue<int> q;

    visited[startNode] = true;

    q.push(startNode);

    cout << "BFS Traversal: ";

    while (!q.empty()) {

        int currentNode = q.front();

        q.pop();

        cout << currentNode << " ";

        for (int neighbor : graph[currentNode]) {

            if (!visited[neighbor]) {

                visited[neighbor] = true;

                q.push(neighbor);

            }

        }

    }

    cout << endl;

}

void printGraph(vector<vector<int>>& graph) {

    cout << "Adjacency List:\n";

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

        if (!graph[i].empty()) {

            cout << i << ": ";

            for (int neighbor : graph[i]) {

                cout << neighbor << " ";

            }

            cout << endl;

        }

    }

}

int main() {

    vector<vector<int>> graph(totalNodes);

    addEdge(graph, 10, 20);

    addEdge(graph, 10, 30);

    addEdge(graph, 20, 40);

    addEdge(graph, 20, 50);

    addEdge(graph, 30, 60);

    printGraph(graph);

    performBFS(graph, 10);

    return 0;

}

**Dfs algorithm**

#include <bits/stdc++.h>

using namespace std;

void addEdge(vector<vector<int>>& graph, int u, int v) {

    graph[u].push\_back(v);

    graph[v].push\_back(u);

}

void performDFS(vector<vector<int>>& graph, int startNode) {

    vector<bool> visited(graph.size(), false);

    stack<int> s;

    s.push(startNode);

    cout << "DFS Traversal: ";

    while (!s.empty()) {

        int currentNode = s.top();

        s.pop();

        if (!visited[currentNode]) {

            visited[currentNode] = true;

            cout << currentNode << " ";

            for (int neighbor : graph[currentNode]) {

                if (!visited[neighbor]) {

                    s.push(neighbor);

                }

            }

        }

    }

    cout << endl;

}

int main() {

    vector<vector<int>> graph(100);

    addEdge(graph, 10, 20);

    addEdge(graph, 10, 30);

    addEdge(graph, 20, 40);

    addEdge(graph, 20, 50);

    addEdge(graph, 30, 60);

    performDFS(graph, 10);

    return 0;

}

**18. Given an n x n binary matrix grid, return the length of the shortest clear path in the matrix. If there is no clear path, return -1. A clear path in a binary matrix is a path from the top-left cell (i.e., (0, 0)) to the bottom-right cell (i.e., (n - 1, n - 1)) such that: All the visited cells of the path are 0. All the adjacent cells of the path are 8-directionally connected (i.e., they are different and they share an edge or a corner).**

#include <iostream>

#include <vector>

#include <queue>

using namespace std;

// Function to check if a cell is within the grid and is clear (0)

bool isValid(int x, int y, int n, vector<vector<int>>& grid) {

return x >= 0 && y >= 0 && x < n && y < n && grid[x][y] == 0;

}

// Function to find the shortest path

int shortestPathBinaryMatrix(vector<vector<int>>& grid) {

int n = grid.size();

// If starting or ending cell is blocked, return -1

if (grid[0][0] != 0 || grid[n - 1][n - 1] != 0) {

return -1;

}

// Directions for moving in 8 directions

vector<pair<int, int>> directions = {

{0, 1}, {1, 0}, {0, -1}, {-1, 0}, {1, 1}, {1, -1}, {-1, 1}, {-1, -1}

};

// BFS queue: stores {x, y, distance}

queue<vector<int>> q;

q.push({0, 0, 1}); // Start from top-left cell with distance 1

grid[0][0] = 1; // Mark the starting cell as visited

while (!q.empty()) {

auto current = q.front();

q.pop();

int x = current[0];

int y = current[1];

int dist = current[2];

// If we reach the bottom-right cell, return the distance

if (x == n - 1 && y == n - 1) {

return dist;

}

// Explore all 8 directions

for (auto dir : directions) {

int newX = x + dir.first;

int newY = y + dir.second;

// If the new cell is valid and clear, visit it

if (isValid(newX, newY, n, grid)) {

q.push({newX, newY, dist + 1});

grid[newX][newY] = 1; // Mark the cell as visited

}

}

}

// If no path is found, return -1

return -1;

}

int main() {

// Example input: binary matrix

vector<vector<int>> grid = {

{0, 1, 0},

{1, 0, 1},

{0, 0, 0}

};

int result = shortestPathBinaryMatrix(grid);

if (result == -1) {

cout << "No clear path found." << endl;

} else {

cout << "The length of the shortest clear path is: " << result << endl;

}

return 0;

}

**19. [Find number of islands](https://www.geeksforgeeks.org/find-number-of-islands/)**

#include <iostream>

#include <vector>

using namespace std;

// Function to perform DFS to mark the current island

void markIsland(vector<vector<int>>& grid, int i, int j) {

int n = grid.size();

int m = grid[0].size();

// Check if the position is out of bounds or water

if (i < 0 || j < 0 || i >= n || j >= m || grid[i][j] == 0) {

return;

}

// Mark the current cell as visited

grid[i][j] = 0;

// Explore the 4 possible directions (up, down, left, right)

markIsland(grid, i + 1, j);

markIsland(grid, i - 1, j);

markIsland(grid, i, j + 1);

markIsland(grid, i, j - 1);

}

// Function to count the number of islands

int countIslands(vector<vector<int>>& grid) {

int n = grid.size();

int m = grid[0].size();

int islandCount = 0;

// Traverse each cell in the grid

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

for (int j = 0; j < m; j++) {

// If a land cell (1) is found, start DFS

if (grid[i][j] == 1) {

islandCount++;

markIsland(grid, i, j); // Mark the entire island as visited

}

}

}

return islandCount;

}

int main() {

// Example input grid

vector<vector<int>> grid = {

{1, 1, 0, 0},

{1, 0, 0, 1},

{0, 0, 1, 1},

{0, 0, 0, 1}

};

cout << "Number of islands: " << countIslands(grid) << endl;

return 0;

}

**20. [Transitive closure of a graph using DFS](https://www.geeksforgeeks.org/transitive-closure-of-a-graph-using-dfs/)**

#include <iostream>

#include <vector>

using namespace std;

// Perform DFS to find all reachable vertices from a given vertex

void dfs(vector<vector<int>>& graph, vector<vector<int>>& closure, int source, int current) {

// Mark the reachability from source to current

closure[source][current] = 1;

// Traverse all neighbors of the current vertex

for (int neighbor = 0; neighbor < graph.size(); neighbor++) {

// If there's an edge and the neighbor has not been visited

if (graph[current][neighbor] == 1 && closure[source][neighbor] == 0) {

dfs(graph, closure, source, neighbor);

}

}

}

// Function to compute the transitive closure of a graph

vector<vector<int>> transitiveClosure(vector<vector<int>>& graph) {

int n = graph.size();

vector<vector<int>> closure(n, vector<int>(n, 0)); // Initialize closure matrix with 0s

// Perform DFS for each vertex to find reachable vertices

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

dfs(graph, closure, i, i);

}

return closure;

}

int main() {

// Example graph represented as an adjacency matrix

vector<vector<int>> graph = {

{1, 1, 0, 0},

{0, 1, 1, 0},

{0, 0, 1, 1},

{0, 0, 0, 1}

};

// Compute the transitive closure

vector<vector<int>> closure = transitiveClosure(graph);

// Print the transitive closure matrix

cout << "Transitive Closure of the given graph:" << endl;

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

for (int j = 0; j < closure[i].size(); j++) {

cout << closure[i][j] << " ";

}

cout << endl;

}

return 0;

}

**21. detect cycle in an undirected graph.**

#include <iostream>

#include <vector>

using namespace std;

// Function to perform DFS to detect a cycle

bool dfs(int node, int parent, vector<vector<int>>& graph, vector<bool>& visited) {

// Mark the current node as visited

visited[node] = true;

// Traverse all adjacent nodes

for (int neighbor : graph[node]) {

// If the neighbor is not visited, perform DFS on it

if (!visited[neighbor]) {

if (dfs(neighbor, node, graph, visited)) {

return true; // Cycle detected

}

}

// If the neighbor is visited and is not the parent, a cycle is detected

else if (neighbor != parent) {

return true;

}

}

return false;

}

// Function to check if the graph contains a cycle

bool hasCycle(int n, vector<vector<int>>& graph) {

vector<bool> visited(n, false); // To keep track of visited nodes

// Perform DFS for each component of the graph

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

if (!visited[i]) {

if (dfs(i, -1, graph, visited)) {

return true; // Cycle detected

}

}

}

return false; // No cycle detected

}

int main() {

// Number of nodes and edges

int n = 5; // Number of vertices (nodes)

vector<vector<int>> graph(n);

// Adding edges to the graph (undirected graph)

graph[0].push\_back(1);

graph[1].push\_back(0);

graph[1].push\_back(2);

graph[2].push\_back(1);

graph[2].push\_back(3);

graph[3].push\_back(2);

graph[3].push\_back(4);

graph[4].push\_back(3);

// Uncomment the next two lines to add a cycle in the graph:

// graph[4].push\_back(1);

// graph[1].push\_back(4);

// Check for a cycle

if (hasCycle(n, graph)) {

cout << "The graph contains a cycle." << endl;

} else {

cout << "The graph does not contain a cycle." << endl;

}

return 0;

}

**22. Find a Mother Vertex in a Graph**

#include <iostream>

#include <vector>

#include <stack>

using namespace std;

// Function to perform DFS on the graph

void dfs(int node, vector<vector<int>>& graph, vector<bool>& visited) {

visited[node] = true;

// Visit all the neighbors of the current node

for (int neighbor : graph[node]) {

if (!visited[neighbor]) {

dfs(neighbor, graph, visited);

}

}

}

// Function to find a mother vertex in the graph

int findMotherVertex(int n, vector<vector<int>>& graph) {

vector<bool> visited(n, false);

int lastVisited = 0;

// Step 1: Perform DFS and find the last finished vertex

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

if (!visited[i]) {

dfs(i, graph, visited);

lastVisited = i;

}

}

// Step 2: Check if the last finished vertex is a mother vertex

fill(visited.begin(), visited.end(), false); // Reset visited array

dfs(lastVisited, graph, visited);

// If all vertices are visited, it's a mother vertex

for (bool isVisited : visited) {

if (!isVisited) {

return -1; // No mother vertex exists

}

}

return lastVisited;

}

int main() {

// Number of vertices

int n = 5;

// Graph represented as an adjacency list

vector<vector<int>> graph = {

{1, 2}, // Edges from vertex 0

{3}, // Edges from vertex 1

{4}, // Edges from vertex 2

{}, // Edges from vertex 3

{} // Edges from vertex 4

};

// Find the mother vertex

int motherVertex = findMotherVertex(n, graph);

if (motherVertex != -1) {

cout << "The mother vertex is: " << motherVertex << endl;

} else {

cout << "There is no mother vertex in the graph." << endl;

}

return 0;

}

23. [Print all path from a given source to a destination](https://www.geeksforgeeks.org/find-paths-given-source-destination/)

#include <iostream>

#include <vector>

using namespace std;

// Helper function to find all paths using DFS

void findAllPaths(vector<vector<int>>& graph, int src, int dest, vector<bool>& visited, vector<int>& path) {

// Mark the current node as visited and add it to the path

visited[src] = true;

path.push\_back(src);

// If the current node is the destination, print the path

if (src == dest) {

for (int node : path) {

cout << node << " ";

}

cout << endl;

} else {

// Recur for all neighbors of the current node

for (int neighbor : graph[src]) {

if (!visited[neighbor]) {

findAllPaths(graph, neighbor, dest, visited, path);

}

}

}

// Backtrack: Remove the current node from the path and mark it as unvisited

path.pop\_back();

visited[src] = false;

}

// Function to print all paths from source to destination

void printAllPaths(vector<vector<int>>& graph, int n, int src, int dest) {

vector<bool> visited(n, false); // To keep track of visited nodes

vector<int> path; // To store the current path

findAllPaths(graph, src, dest, visited, path);

}

int main() {

// Number of vertices

int n = 5;

// Graph represented as an adjacency list

vector<vector<int>> graph = {

{1, 2}, // Edges from vertex 0

{3}, // Edges from vertex 1

{3, 4}, // Edges from vertex 2

{4}, // Edges from vertex 3

{} // Edges from vertex 4

};

// Source and destination

int src = 0, dest = 4;

// Print all paths from source to destination

cout << "All paths from " << src << " to " << dest << " are:" << endl;

printAllPaths(graph, n, src, dest);

return 0;

}

25. **Find next Smaller of next Greater in an array**

#include <iostream>

#include <vector>

#include <stack>

using namespace std;

// Function to find the Next Greater Element (NGE) and

// the Next Smaller Element (NSE) of the NGE

vector<int> nextSmallerOfNextGreater(vector<int>& arr) {

int n = arr.size();

stack<int> s; // Stack for storing elements

vector<int> nge(n, -1); // Store next greater element

vector<int> nse(n, -1); // Store next smaller element of NGE

// Step 1: Find the Next Greater Element for each element

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

while (!s.empty() && arr[s.top()] < arr[i]) {

nge[s.top()] = arr[i]; // Assign NGE for the element at top of the stack

s.pop();

}

s.push(i); // Push current element index onto the stack

}

// Step 2: Find the Next Smaller Element for each NGE found

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

if (nge[i] != -1) { // Only if NGE is found

while (!s.empty() && arr[s.top()] > nge[i]) {

nse[s.top()] = arr[s.top()]; // Assign NSE for the NGE

s.pop();

}

}

}

return nse;

}

int main() {

vector<int> arr = {4, 5, 2, 10, 8};

// Find the Next Smaller of Next Greater for each element

vector<int> result = nextSmallerOfNextGreater(arr);

// Print the result

cout << "Next Smaller of Next Greater for each element: ";

for (int i : result) {

cout << i << " ";

}

cout << endl;

return 0;

}

**26. [Implement two stacks in an array](https://www.geeksforgeeks.org/implement-two-stacks-in-an-array/" \t "_blank)**

#include <iostream>

using namespace std;

class TwoStacks {

public:

int \*arr; // Array to store elements

int size; // Size of the array

int top1, top2; // Indices for the two stacks

// Constructor to initialize the two stacks

TwoStacks(int n) {

size = n;

arr = new int[size];

top1 = -1; // Stack 1 starts from the left

top2 = size; // Stack 2 starts from the right

}

// Function to push an element into the first stack

void push1(int x) {

// Check if there is space available

if (top1 < top2 - 1) {

top1++;

arr[top1] = x;

} else {

cout << "Overflow in Stack 1\n";

}

}

// Function to push an element into the second stack

void push2(int x) {

// Check if there is space available

if (top1 < top2 - 1) {

top2--;

arr[top2] = x;

} else {

cout << "Overflow in Stack 2\n";

}

}

// Function to pop an element from the first stack

int pop1() {

// Check if stack 1 is empty

if (top1 == -1) {

cout << "Underflow in Stack 1\n";

return -1; // Return -1 if stack 1 is empty

} else {

int popped = arr[top1];

top1--;

return popped;

}

}

// Function to pop an element from the second stack

int pop2() {

// Check if stack 2 is empty

if (top2 == size) {

cout << "Underflow in Stack 2\n";

return -1; // Return -1 if stack 2 is empty

} else {

int popped = arr[top2];

top2++;

return popped;

}

}

// Function to check if stack 1 is empty

bool isEmpty1() {

return top1 == -1;

}

// Function to check if stack 2 is empty

bool isEmpty2() {

return top2 == size;

}

};

int main() {

// Create a TwoStacks object with a size of 10

TwoStacks ts(10);

// Push elements into the first stack

ts.push1(5);

ts.push1(10);

ts.push1(15);

// Push elements into the second stack

ts.push2(20);

ts.push2(25);

ts.push2(30);

// Pop elements from the first stack

cout << "Popped from Stack 1: " << ts.pop1() << endl;

cout << "Popped from Stack 1: " << ts.pop1() << endl;

// Pop elements from the second stack

cout << "Popped from Stack 2: " << ts.pop2() << endl;

cout << "Popped from Stack 2: " << ts.pop2() << endl;

return 0;

}

**27. [Evaluate Postfix Expression](https://www.geeksforgeeks.org/stack-set-4-evaluation-postfix-expression/" \t "_blank)**

#include <bits/stdc++.h>

#define MAX\_SIZE 100

int stack[MAX\_SIZE];

int top = -1;

void push(int item) {

    if (top >= MAX\_SIZE - 1) {

        printf("Stack Overflow\n");

        return;

    }

    top++;

    stack[top] = item;

}

int pop() {

    if (top < 0) {

        printf("Stack Underflow\n");

        return -1;

    }

    int item = stack[top];

    top--;

    return item;

}

int is\_operator(char symbol) {

    if (symbol == '+' || symbol == '-' || symbol == '\*' || symbol == '/') {

        return 1;

    }

    return 0;

}

int evaluate(char\* expression) {

    int i = 0;

    char symbol = expression[i];

    int operand1, operand2, result;

    while (symbol != '\0') {

        if (symbol >= '0' && symbol <= '9') {

            int num = symbol - '0';

            push(num);

        } else if (is\_operator(symbol)) {

            operand2 = pop();

            operand1 = pop();

            switch (symbol) {

                case '+':

                    result = operand1 + operand2;

                    break;

                case '-':

                    result = operand1 - operand2;

                    break;

                case '\*':

                    result = operand1 \* operand2;

                    break;

                case '/':

                    result = operand1 / operand2;

                    break;

            }

            push(result);

        }

        i++;

        symbol = expression[i];

    }

    result = pop();

    return result;

}

int main() {

    char expression[] = "5 6 7 + \* 8 -";

    int result = evaluate(expression);

    printf("Result= %d\n", result);

    return 0;

}

**28. [Next Greater Element](https://www.geeksforgeeks.org/next-greater-element/)**

#include <iostream>

#include <stack>

#include <vector>

using namespace std;

// Function to find the Next Greater Element for each element in the array

void nextGreaterElement(vector<int>& arr) {

stack<int> s; // Stack to store elements

vector<int> result(arr.size(), -1); // Result vector initialized to -1

// Traverse the array

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

// While stack is not empty and the current element is greater than the stack's top element

while (!s.empty() && arr[i] > arr[s.top()]) {

result[s.top()] = arr[i]; // Current element is the NGE for stack's top element

s.pop(); // Remove the top element from the stack

}

s.push(i); // Push the index of the current element onto the stack

}

// Print the result (Next Greater Element for each element)

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

cout << "Next Greater Element for " << arr[i] << " is: " << result[i] << endl;

}

}

int main() {

vector<int> arr = {4, 5, 2, 10, 8};

// Find and print the Next Greater Element for each element in the array

nextGreaterElement(arr);

return 0;

}

**29. [Nearest Smaller Element](https://www.geeksforgeeks.org/find-the-nearest-smaller-numbers-on-left-side-in-an-array/" \t "_blank)**

#include <iostream>

#include <stack>

#include <vector>

using namespace std;

// Function to find the Nearest Smaller Element for each element in the array

void nearestSmallerElement(vector<int>& arr) {

stack<int> s; // Stack to store elements

vector<int> result(arr.size(), -1); // Result vector initialized to -1

// Traverse the array

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

// While stack is not empty and the top element of the stack is not smaller than the current element

while (!s.empty() && s.top() >= arr[i]) {

s.pop(); // Pop elements from the stack that are not smaller than the current element

}

// If stack is not empty, the top element is the Nearest Smaller Element

if (!s.empty()) {

result[i] = s.top();

}

// Push the current element onto the stack

s.push(arr[i]);

}

// Print the result (Nearest Smaller Element for each element)

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

cout << "Nearest Smaller Element for " << arr[i] << " is: " << result[i] << endl;

}

}

int main() {

vector<int> arr = {4, 5, 2, 10, 8};

// Find and print the Nearest Smaller Element for each element in the array

nearestSmallerElement(arr);

return 0;

}

**30. Sort a stack using a temporary stack**

#include <iostream>

#include <stack>

using namespace std;

// Function to sort the stack using a temporary stack

void sortedInsert(stack<int>& tempStack, int element) {

// If temporary stack is empty or the element is greater than the top of the stack

if (tempStack.empty() || tempStack.top() <= element) {

tempStack.push(element);

} else {

// Pop from temporary stack and push back into main stack until correct position is found

int topElement = tempStack.top();

tempStack.pop();

sortedInsert(tempStack, element); // Recursively insert the element

tempStack.push(topElement); // Push the popped element back into the temporary stack

}

}

// Function to sort the stack using recursion

void sortStack(stack<int>& s) {

if (!s.empty()) {

// Pop an element from the main stack

int temp = s.top();

s.pop();

// Sort the remaining stack

sortStack(s);

// Insert the popped element in sorted order into the temporary stack

sortedInsert(s, temp);

}

}

// Function to print the stack

void printStack(stack<int> s) {

while (!s.empty()) {

cout << s.top() << " ";

s.pop();

}

cout << endl;

}

int main() {

stack<int> s;

// Pushing elements into the stack

s.push(30);

s.push(10);

s.push(20);

s.push(50);

s.push(40);

cout << "Original Stack: ";

printStack(s);

// Sorting the stack

sortStack(s);

cout << "Sorted Stack: ";

printStack(s);

return 0;

}

**31. [Reverse a stack using recursion](https://www.geeksforgeeks.org/reverse-a-stack-using-recursion/" \t "_blank)**

#include <iostream>

#include <stack>

using namespace std;

// Function to insert an element at the bottom of the stack

void insertAtBottom(stack<int>& s, int element) {

// Base case: If the stack is empty, push the element

if (s.empty()) {

s.push(element);

} else {

// Recursive case: pop the top element, recursively insert the element at the bottom, then push the popped element back

int topElement = s.top();

s.pop();

insertAtBottom(s, element); // Insert element at the bottom

s.push(topElement); // Push the top element back after inserting the element at the bottom

}

}

// Function to reverse the stack using recursion

void reverseStack(stack<int>& s) {

// Base case: If the stack is empty, do nothing

if (!s.empty()) {

// Pop the top element

int topElement = s.top();

s.pop();

// Recursively reverse the remaining stack

reverseStack(s);

// Insert the popped element at the bottom of the stack

insertAtBottom(s, topElement);

}

}

// Function to print the stack

void printStack(stack<int> s) {

while (!s.empty()) {

cout << s.top() << " ";

s.pop();

}

cout << endl;

}

int main() {

stack<int> s;

// Pushing elements into the stack

s.push(10);

s.push(20);

s.push(30);

s.push(40);

s.push(50);

cout << "Original Stack: ";

printStack(s);

// Reversing the stack

reverseStack(s);

cout << "Reversed Stack: ";

printStack(s);

return 0;

}

# **Infix to postfix**

#include <bits/stdc++.h>

using namespace std;

#define MAX\_SIZE 100

char s[MAX\_SIZE];

int top = -1;

void push(char c) {

    if (top == MAX\_SIZE - 1) {

        cout << "Stack Overflow" << endl;

        return;

    }

    s[++top] = c;

}

char pop() {

    if (top == -1) {

        cout << "Stack Underflow" << endl;

        return 0;

    }

    return s[top--];

}

char peek() {

    if (top == -1) {

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

        return 0;

    }

    return s[top];

}

bool isEmpty() {

    return top == -1;

}

int precedence(char op) {

    if (op == '+' || op == '-')

        return 1;

    if (op == '\*' || op == '/')

        return 2;

    if (op == '^')

        return 3;

    return 0;

}

bool isOperator(char c) {

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

}

bool isOperand(char c) {

    return (c >= 'a' && c <= 'z') || (c >= 'A' && c <= 'Z') || (c >= '0' && c <= '9');

}

string infixToPostfix(string infix) {

    string postfix = "";

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

        char c = infix[i];

        if (isOperand(c)) {

            postfix += c;

        } else if (c == '(') {

            push(c);

        } else if (c == ')') {

            while (!isEmpty() && peek() != '(') {

                postfix += pop();

            }

            if (!isEmpty() && peek() == '(') {

                pop();

            }

        } else if (isOperator(c)) {

            while (!isEmpty() && precedence(peek()) >= precedence(c)) {

                postfix += pop();

            }

            push(c);

        }

    }

    while (!isEmpty()) {

        postfix += pop();

    }

    return postfix;

}

int main() {

    string infix;

    cout << "Enter infix expression: ";

    cin >> infix;

    string postfix = infixToPostfix(infix);

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

    return 0;

}

# DFS directed graph:

#include <bits/stdc++.h>

using namespace std;

bool isCyclicDFS(int node, vector<vector<int>>& graph, vector<bool>& visited, vector<bool>& recStack) {

    visited[node] = true;

    recStack[node] = true;

    for (int neighbor : graph[node]) {

        if (!visited[neighbor]) {

            if (isCyclicDFS(neighbor, graph, visited, recStack)) {

                return true;

            }

        } else if (recStack[neighbor]) {

            return true;

        }

    }

    recStack[node] = false;

    return false;

}

bool detectCycleDFS(vector<vector<int>>& graph, int totalNodes) {

    vector<bool> visited(totalNodes, false);

    vector<bool> recStack(totalNodes, false);

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

        if (!visited[i]) {

            if (isCyclicDFS(i, graph, visited, recStack)) {

                return true;

            }

        }

    }

    return false;

}

int main() {

    int totalNodes = 4;

    vector<vector<int>> graph(totalNodes);

    graph[0].push\_back(1);

    graph[1].push\_back(2);

    graph[2].push\_back(3);

    graph[3].push\_back(1);

    if (detectCycleDFS(graph, totalNodes)) {

        cout << "Cycle detected using DFS\n";

    } else {

        cout << "No cycle detected using DFS\n";

    }

    return 0;

}