1. Design a stack data structure using two queues. Implement the push, pop and top operations for the stack.

**SOURCE CODE**

#include <stdio.h>

#include <stdlib.h>

// Queue structure

struct Queue {

int front, rear, capacity;

int\* array;

};

// Stack structure

struct Stack {

struct Queue\* q1;

struct Queue\* q2;

};

// Function to create a new queue

struct Queue\* createQueue(int capacity) {

struct Queue\* queue = (struct Queue\*)malloc(sizeof(struct Queue));

queue->capacity = capacity;

queue->front = queue->rear = -1;

queue->array = (int\*)malloc(queue->capacity \* sizeof(int));

return queue;

}

// Function to enqueue an element

void enqueue(struct Queue\* queue, int item) {

if (queue->rear == queue->capacity - 1) {

printf("Queue is full.\n");

return;

}

queue->array[++queue->rear] = item;

if (queue->front == -1) {

queue->front = 0;

}

}

// Function to dequeue an element

int dequeue(struct Queue\* queue) {

if (queue->front == -1) {

printf("Queue is empty.\n");

return -1;

}

int item = queue->array[queue->front];

if (queue->front == queue->rear) {

queue->front = queue->rear = -1;

} else {

queue->front++;

}

return item;

}

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

enqueue(stack->q1, item);

}

int pop(struct Stack\* stack) {

if (stack->q1->front == -1) {

printf("Stack is empty.\n");

return -1;

}

while (stack->q1->front != stack->q1->rear) {

enqueue(stack->q2, dequeue(stack->q1));

}

int popped = dequeue(stack->q1);

struct Queue\* temp = stack->q1;

stack->q1 = stack->q2;

stack->q2 = temp;

return popped;

}

int top(struct Stack\* stack) {

if (stack->q1->front == -1) {

printf("Stack is empty.\n");

return -1;

}

while (stack->q1->front != stack->q1->rear) {

enqueue(stack->q2, dequeue(stack->q1));

}

int topElement = dequeue(stack->q1);

enqueue(stack->q2, topElement);

struct Queue\* temp = stack->q1;

stack->q1 = stack->q2;

stack->q2 = temp;

return topElement;

}

int main() {

struct Stack\* stack = (struct Stack\*)malloc(sizeof(struct Stack));

stack->q1 = createQueue(100);

stack->q2 = createQueue(100);

push(stack, 10);

push(stack, 20);

push(stack, 30);

printf("Top element: %d\n", top(stack));

printf("Popped element: %d\n", pop(stack));

printf("Top element: %d\n", top(stack));

return 0;

}

**TIME COMPLEXITY**

The push operation is efficient with a time complexity of O(1), but the pop and top operations have a time complexity of O(n) due to the need to move elements between the two queues. This is not the most efficient way to implement a stack using queues in terms of time complexity. There are alternative methods, such as using a single queue and reversing its elements, which can achieve O(1) time complexity for all operations.

**OUTPUT**

Top element: 30

Popped element: 30

Top element: 20

1. Design a queue data structure using two stacks. Implement the enqueue, dequeue and front operations for the queue.

**SOURCE CODE**

#include <stdio.h>

#include <stdlib.h>

#define MAX\_SIZE 100

// Stack structure

struct Stack {

int top;

int array[MAX\_SIZE];

};

// Queue structure

struct Queue {

struct Stack\* s1; // For enqueue

struct Stack\* s2; // For dequeue

};

// Function to create a new stack

struct Stack\* createStack() {

struct Stack\* stack = (struct Stack\*)malloc(sizeof(struct Stack));

stack->top = -1;

return stack;

}

// Function to push an element onto the stack

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

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

printf("Stack is full.\n");

return;

}

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

}

// Function to pop the top element from the stack

int pop(struct Stack\* stack) {

if (stack->top == -1) {

printf("Stack is empty.\n");

return -1;

}

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

}

// Function to create a new queue

struct Queue\* createQueue() {

struct Queue\* queue = (struct Queue\*)malloc(sizeof(struct Queue));

queue->s1 = createStack();

queue->s2 = createStack();

return queue;

}

// Function to enqueue an element

void enqueue(struct Queue\* queue, int item) {

while (queue->s1->top != -1) {

push(queue->s2, pop(queue->s1));

}

push(queue->s1, item);

while (queue->s2->top != -1) {

push(queue->s1, pop(queue->s2));

}

}

// Function to dequeue an element

int dequeue(struct Queue\* queue) {

if (queue->s1->top == -1) {

printf("Queue is empty.\n");

return -1;

}

return pop(queue->s1);

}

// Function to get the front element of the queue

int front(struct Queue\* queue) {

if (queue->s1->top == -1) {

printf("Queue is empty.\n");

return -1;

}

return queue->s1->array[queue->s1->top];

}

int main() {

struct Queue\* queue = createQueue();

enqueue(queue, 10);

enqueue(queue, 20);

enqueue(queue, 30);

printf("Front element: %d\n", front(queue));

printf("Dequeued element: %d\n", dequeue(queue));

printf("Front element: %d\n", front(queue));

return 0;

}

**TIME COMPLEXITY**

In this implementation, while the dequeue and front operations are efficient with a time complexity of O(1), the enqueue operation is not as efficient, with a time complexity of O(n) in the worst case. This is due to the need to move elements between the two stacks during the enqueue operation. There are more efficient ways to implement a queue using two stacks, such as using a second stack to store elements in reverse order and transferring elements only when necessary, achieving O(1) time complexity for all operations.

**OUTPUT**

Front element: 10

Dequeued element: 10

Front element: 20

1. Write a program to reverse the order of the elements in a given queue using a stack. The program should take a queue as input and output the reversed queue.

**SOURCE CODE**

#include <stdio.h>

#include <stdlib.h>

#define MAX\_SIZE 100

// Queue structure

struct Queue {

int front, rear, size;

int array[MAX\_SIZE];

};

// Stack structure

struct Stack {

int top;

int array[MAX\_SIZE];

};

// Function to create a new queue

struct Queue\* createQueue() {

struct Queue\* queue = (struct Queue\*)malloc(sizeof(struct Queue));

queue->front = queue->rear = -1;

queue->size = 0;

return queue;

}

// Function to create a new stack

struct Stack\* createStack() {

struct Stack\* stack = (struct Stack\*)malloc(sizeof(struct Stack));

stack->top = -1;

return stack;

}

// Function to enqueue an element

void enqueue(struct Queue\* queue, int item) {

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

printf("Queue is full.\n");

return;

}

queue->array[++queue->rear] = item;

queue->size++;

if (queue->front == -1) {

queue->front = 0;

}

}

// Function to dequeue an element

int dequeue(struct Queue\* queue) {

if (queue->front == -1) {

printf("Queue is empty.\n");

return -1;

}

int item = queue->array[queue->front++];

queue->size--;

if (queue->front > queue->rear) {

queue->front = queue->rear = -1;

}

return item;

}

// Function to push an element onto the stack

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

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

printf("Stack is full.\n");

return;

}

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

}

// Function to pop the top element from the stack

int pop(struct Stack\* stack) {

if (stack->top == -1) {

printf("Stack is empty.\n");

return -1;

}

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

}

// Function to reverse the order of elements in a queue

void reverseQueue(struct Queue\* queue) {

struct Stack\* stack = createStack();

while (queue->size > 0) {

push(stack, dequeue(queue));

}

while (stack->top != -1) {

enqueue(queue, pop(stack));

}

}

// Function to print the queue

void printQueue(struct Queue\* queue) {

printf("Reversed Queue: ");

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

printf("%d ", queue->array[i]);

}

printf("\n");

}

int main() {

struct Queue\* queue = createQueue();

enqueue(queue, 10);

enqueue(queue, 20);

enqueue(queue, 30);

enqueue(queue, 40);

printf("Original Queue: ");

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

printf("%d ", queue->array[i]);

}

printf("\n");

reverseQueue(queue);

printQueue(queue);

return 0;

}

**TIME COMPLEXITY**

The most time-consuming operation in the provided program is the reverseQueue function, which has a time complexity of O(n), where n is the number of elements in the queue. Other operations, such as enqueue, dequeue, push, and pop, have a time complexity of O(1).

**OUTPUT**

Original Queue: 10 20 30 40

Reversed Queue: 40 30 20 10

1. You are given a string containing parentheses, curly brackets and square brackets. Design a program to check whether the given string has balanced parentheses. The string is considered balanced if each opening parenthesis is closed by the correct corresponding closing parenthesis.

**SOURCE CODE**

#include <stdio.h>

#include <stdlib.h>

#include <stdbool.h>

#define MAX\_SIZE 100

struct Stack {

int top;

char array[MAX\_SIZE];

};

struct Stack\* createStack() {

struct Stack\* stack = (struct Stack\*)malloc(sizeof(struct Stack));

stack->top = -1;

return stack;

}

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

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

printf("Stack is full.\n");

return;

}

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

}

char pop(struct Stack\* stack) {

if (stack->top == -1) {

printf("Stack is empty.\n");

return '\0'; }

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

}

bool isOpening(char c) {

return (c == '(' || c == '{' || c == '[');

}

bool isClosing(char c) {

return (c == ')' || c == '}' || c == ']');

}

bool isPair(char opening, char closing) {

return ((opening == '(' && closing == ')') ||

(opening == '{' && closing == '}') ||

(opening == '[' && closing == ']'));

}

bool isBalanced(char\* str) {

struct Stack\* stack = createStack();

for (int i = 0; str[i] != '\0'; i++) {

if (isOpening(str[i])) {

push(stack, str[i]);

} else if (isClosing(str[i])) {

if (stack->top == -1 || !isPair(pop(stack), str[i])) {

return false;

}

}

}

return (stack->top == -1);

}

int main() {

char str[MAX\_SIZE];

printf("Enter a string: ");

scanf("%s", str);

if (isBalanced(str)) {

printf("The string has balanced parentheses.\n");

} else {

printf("The string does not have balanced parentheses.\n");

}

return 0;

}

**TIME COMPLEXITY**

The most time-consuming operation in the program is the “isBalanced” function, which iterates through the input string and performs constant-time operations for each character. Other operations, such as push and pop, have a time complexity of O(1).

**OUTPUT**

Enter a string: {Fruits[Mango]}

The string has balanced parentheses.

Enter a string: {Tree[Banyan)}

The string does not have balanced parentheses.

1. Implement a function to convert an infix expression to postfix expression using stacks. The infix expression contains operands, operators and parentheses. The output should be a postfix expression with the same order of operations as the original expression.

**SOURCE CODE**

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <ctype.h>

#define MAX\_SIZE 100

// Stack structure

struct Stack {

int top;

char array[MAX\_SIZE];

};

// Function to create a new stack

struct Stack\* createStack() {

struct Stack\* stack = (struct Stack\*)malloc(sizeof(struct Stack));

stack->top = -1;

return stack;

}

// Function to push an element onto the stack

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

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

printf("Stack is full.\n");

return;

}

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

}

// Function to pop the top element from the stack

char pop(struct Stack\* stack) {

if (stack->top == -1) {

printf("Stack is empty.\n");

return '\0';

}

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

}

// Function to get the top element from the stack without popping

char peek(struct Stack\* stack) {

if (stack->top == -1) {

return '\0';

}

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

}

// Function to determine the precedence of an operator

int precedence(char op) {

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

return 1;

} else if (op == '\*' || op == '/') {

return 2;

}

return 0;

}

// Function to convert infix expression to postfix expression

void infixToPostfix(char\* infix, char\* postfix) {

struct Stack\* stack = createStack();

int i, j;

i = j = 0;

while (infix[i] != '\0') {

char c = infix[i];

if (isalnum(c)) {

postfix[j++] = c;

} else if (c == '(') {

push(stack, c);

} else if (c == ')') {

while (peek(stack) != '(') {

postfix[j++] = pop(stack);

}

pop(stack); // Pop '('

} else {

while (precedence(c) <= precedence(peek(stack))) {

postfix[j++] = pop(stack);

}

push(stack, c);

}

i++;

}

while (stack->top != -1) {

postfix[j++] = pop(stack);

}

postfix[j] = '\0';

}

int main() {

char infix[MAX\_SIZE];

printf("Enter an infix expression: ");

scanf("%s", infix);

char postfix[MAX\_SIZE];

infixToPostfix(infix, postfix);

printf("Postfix expression: %s\n", postfix);

return 0;

}

**TIME COMPLEXITY**

The most time-consuming operation in the program is the “infixToPostfix” function, which iterates through the input infix expression and performs constant-time operations for each character. Other operations, such as push, pop, peek, and precedence determination, have a time complexity of O(1).

**OUTPUT**

Enter an infix expression: (2+3)\*(4+5)

Postfix expression: 23+45+\*

1. Design a function to evaluate a postfix expression using a stack. The postfix expression contains operands and operators. The function should return the result of the expression.

**SOURCE CODE**

#include <stdio.h>

#include <stdlib.h>

#include <ctype.h>

#include <stdbool.h>

#define MAX\_SIZE 100

// Stack structure

struct Stack {

int top;

int array[MAX\_SIZE];

};

// Function to create a new stack

struct Stack\* createStack() {

struct Stack\* stack = (struct Stack\*)malloc(sizeof(struct Stack));

stack->top = -1;

return stack;

}

// Function to push an element onto the stack

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

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

printf("Stack is full.\n");

return;

}

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

}

// Function to pop the top element from the stack

int pop(struct Stack\* stack) {

if (stack->top == -1) {

printf("Stack is empty.\n");

return -1;

}

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

}

// Function to check if a character is an operand

bool isOperand(char c) {

return isdigit(c);

}

// Function to evaluate a postfix expression

int evaluatePostfix(char\* postfix) {

struct Stack\* stack = createStack();

int i = 0;

while (postfix[i] != '\0') {

char c = postfix[i];

if (isOperand(c)) {

push(stack, c - '0'); // Convert char to integer

} else {

int operand2 = pop(stack);

int operand1 = pop(stack);

switch (c) {

case '+':

push(stack, operand1 + operand2);

break;

case '-':

push(stack, operand1 - operand2);

break;

case '\*':

push(stack, operand1 \* operand2);

break;

case '/':

push(stack, operand1 / operand2);

break;

}

}

i++;

}

return pop(stack);

}

int main() {

char postfix[MAX\_SIZE];

printf("Enter a postfix expression: ");

scanf("%s", postfix);

int result = evaluatePostfix(postfix);

printf("Result: %d\n", result);

return 0;

}

**­TIME COMPLEXITY**

Overall, the program has a time complexity of O(n), where n is the length of the postfix expression. This makes the program relatively efficient for evaluating postfix expressions.

**OUTPUT**

Enter a postfix expression: 52+3\*

Result: 21

1. Given an array of integers, design a function to find the next greater element for each element in the array. The next greater element for an element for an element is the first greater element to its right. If no greater element exists, output -1 for that element.

**SOURCE CODE**

#include <stdio.h>

#define MAX\_SIZE 100

// Function to find the next greater element for each element in the array

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

int result[n]; // To store the next greater elements

int stack[MAX\_SIZE]; // Stack to hold indices

int top = -1; // Initialize stack top

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

// Keep popping elements from the stack while they are smaller than the current element

while (top >= 0 && arr[i] > arr[stack[top]]) {

result[stack[top--]] = arr[i];

}

// Push the current index onto the stack

stack[++top] = i;

}

// The remaining elements in the stack have no greater elements to their right

while (top >= 0) {

result[stack[top--]] = -1;

}

// Print the next greater elements for each element

printf("Next Greater Elements:\n");

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

printf("%d -> %d\n", arr[i], result[i]);

}

}

int main() {

int n;

printf("Enter the number of elements: ");

scanf("%d", &n);

int arr[MAX\_SIZE];

printf("Enter the array elements: ");

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

scanf("%d", &arr[i]);

}

findNextGreater(arr, n);

return 0;

}

**TIME COMPLEXITY**

The program has a linear time complexity of O(n), where n is the number of elements in the input array. This makes the program efficient for finding the next greater element for each element.

**OUTPUT**

Enter the number of elements: 5

Enter the array elements: 12 34 10 45 57

Next Greater Elements:

12 -> 34

34 -> 45

10 -> 45

45 -> 57

57 -> -1