Data Structures: Stacks and Queues in C

Welcome, aspiring C programmers! Today, we'll unravel the fundamental concepts of Stacks and Queues, two essential linear data structures. Understanding their array-based implementations in C is crucial for efficient program design and problem-solving.

Agenda

What We'll Cover Today

01

Introduction to Stacks

Defining LIFO and its applications.

02

Stack Operations in C

Push, Pop, Peek, and isEmpty/isFull implementations, Applications.

03

Introduction to Queues

Defining FIFO and its real-world relevance.

04

Queue Operations in C

Enqueue, Dequeue, Front, and Rear implementations.

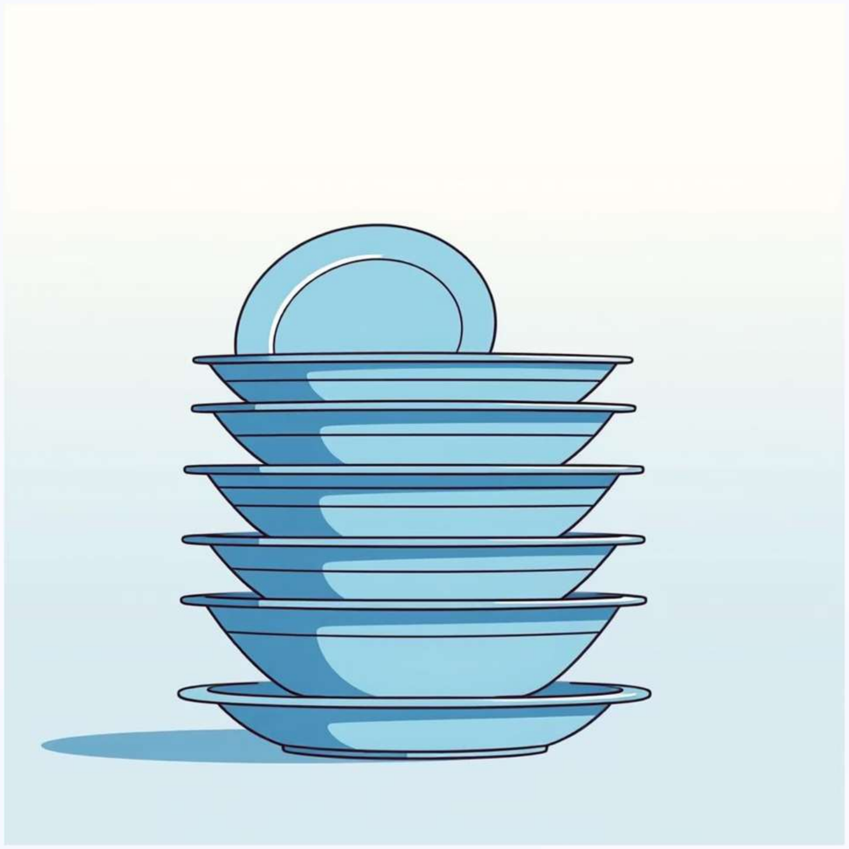
05

Key Takeaways & Best Practices

Summarizing concepts and optimizing your code.

Understanding the Stack: Last-In, First-Out (LIFO)

Imagine a stack of plates: you always add new plates to the top, and you always remove plates from the top. The last plate you put on is the first one you take off. This "Last-In, First-Out" (LIFO) principle defines a stack data structure.



Key Characteristics:

- Items are added and removed from the same end, called the "top".
- Commonly used in function call management, expression evaluation, and undo/redo features.

Array-Based Stack Operations in C

Implementing a stack using an array requires a fixed size and a variable to keep track of the "top" element's index.

1

Push: Adding an Element

Adds an element to the top of the stack. Before pushing, always check if the stack is full to prevent overflow.

`void push(int item)`

2

Pop: Removing an Element

Removes and returns the element from the top of the stack. Check if the stack is empty to avoid underflow.

`int pop()`

3

Peek/Top: View Top Element

Returns the top element without removing it. Essential for checking the next item to be processed.

`int peek()`

4

isEmpty & isFull

Utility functions to check the current state of the stack before performing push or pop operations.

`bool isEmpty()`bool isFull()

Deep Dive: Example Stack Code

```
#define MAX_SIZE 10
int stack[MAX_SIZE];
int top = -1;
bool isEmpty() { return top == -1; }
bool isFull() { return top == MAX_SIZE - 1; }

void push(int item) {
    if (isFull()) { /* Handle overflow */ }
    else { stack[++top] = item; }
}

int pop() {
    if (isEmpty()) { /* Handle underflow */ }
    else { return stack[top--]; }
}

int peek() {
    if (isEmpty()) { /* Handle empty */ }
    else { return stack[top]; }
}
```

This snippet illustrates the core logic for stack operations. Remember to include robust error handling for real-world applications!

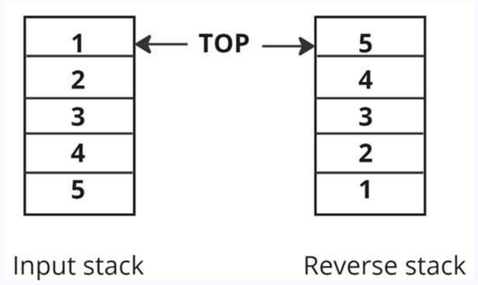
Applications of Stacks

- Reversing a list
- Parentheses checker
- Conversion of an infix expression into a postfix expression
- Evaluation of a postfix expression
- Conversion of an infix expression into a prefix expression
- Evaluation of a prefix expression
- Recursion
- Tower of Hanoi

Applications of Stacks

Reversing a List

• A list of numbers/characters can be reversed by reading each number/character from an array starting from the first index and pushing it on a stack.
• Once all the numbers have been read, the numbers can be popped one at a time and then stored in the array starting from the first index.



```
STEP 1 - While `InputStack` is not empty:
        - Pop the top element from `InputStack`.
        - Push the popped element onto `ReverseStack`.

STEP 2 - Create an empty list called `ReversedList`.

STEP 3 - While `ReverseStack` is not empty:
        - Pop the top element from `ReverseStack`.
        - Append the popped element to `ReversedList`.
```

The `ReversedList` now contains the reversed elements of the original list.

Introducing the Queue: First-In, First-Out (FIFO)

Think of a line at a coffee shop: the first person to join the line is the first person to be served. This "First-In, First-Out" (FIFO) principle is the hallmark of a queue data structure.

Key Characteristics:

- Items are added at one end (rear) and removed from the other (front).
- Ideal for task scheduling, print queues, and breadth-first search algorithms.

Array-Based Queue Operations in C

Implementing a queue with an array is slightly more complex than a stack, as you need to manage both a 'front' and 'rear' pointer.

1

Enqueue: Adding to Queue

Adds an element to the rear of the queue. Check for queue full conditions before adding.

`void enqueue(int item)`

2

Dequeue: Removing from Queue

Removes and returns the element from the front of the queue. Verify the queue isn't empty first.

`int dequeue()`

3

Front & Rear

These functions return the elements at the front and rear of the queue, respectively, without removing them.

`int front()int rear()`

4

isEmpty & isFull

Critical for preventing errors and managing the queue's state, similar to stacks.

`bool isEmpty()`bool isFull()

Example Queue Implementation

```c
#define MAX_SIZE 10
int queue[MAX_SIZE];
int front = -1, rear = -1;

bool isEmpty() { return front == -1; }
bool isFull() { return (rear + 1) % MAX_SIZE == front; }

void enqueue(int item) {
    if (isFull()) { /* Handle overflow */ }
    else {
        if (isEmpty()) front = 0;
        rear = (rear + 1) % MAX_SIZE;
        queue[rear] = item;
    }
}

int dequeue() {
    if (isEmpty()) { /* Handle underflow */ }
    else {
        int item = queue[front];
        if (front == rear) front = rear = -1;
        else front = (front + 1) % MAX_SIZE;
        return item;
    }
}
```

This circular array implementation efficiently reuses space after dequeue operations. Managing `front` and `rear` pointers is key.

Key Takeaways & Best Practices

Efficiency is Key

Array-based implementations offer simplicity and direct memory access, making them efficient for many use cases.

Manage Edge Cases

Always implement checks for `isEmpty()` and `isFull()` to prevent runtime errors like overflow or underflow.

Choose Wisely

Select Stack (LIFO) or Queue (FIFO) based on your application's data handling requirements. Understand their fundamental differences.

Exercise

1. Create an algorithm that enables a user to undo and redo last conducted activities by using stack(s).
    - You are given a stack of actions.
    - Your task is to think of a way to implement a functionality to undo and redo the action is the given stack. You are allowed to use any other data structure you have learned so far. (array, stack)
    - Write the pseudocode for undo(), redo(). You can assume that push(),pop() and other base functions are already implemented.

Exercise

2.Write a program to convert decimal numbers to binary using a stack.

- Assume modulo operator (%) give you the remainder and '/' gives the quotient
- Example 11% 2 = 1 and 11/2 = 5

Exercise

3. Write a program that reads a postfix expression and evaluates it.
   Use a stack of numbers. Read one term at the time:
   if the term is a number, push it onto the stack
   if the term is an operator, pop the two topmost numbers, apply the operator to them, and push the result back onto the stack

   examples of post fix expressions :
   6 8 + 1 2 + *
   4 5 5 6 * - +

Exercise

4. Write a C program to generate binary numbers between 0 to n using a queue

For example , for N=10, the binary numbers are
1 10 11 100 101 110 111 1000 1001 1010 10111100 1101 1110
1111 10000

Thank You!