Stacks and Queues

Data Structures and Algorithms Tutorial

IIITH - S26

LIFO and FIFO Data Structures

Agenda

1. Overview
2. Stacks
3. Queues
4. Practice Problems

Overview

What are Stacks and Queues?

Definition

- Linear data structures with specific access patterns
- Abstract Data Types (ADTs) - define behavior, not implementation

Stack

- LIFO: Last In First Out
- Think: Stack of plates, browser back button, undo operation

Queue

- FIFO: First In First Out
- Think: Line at a store, printer queue, task scheduling

Stacks

Stack: Main Idea

Core Concept

- Add elements at one end
- Remove elements from same end
- All operations at ONE end

Key Operations

- push(x): Add element
- pop(): Remove and return element
- front(): Return element without removing
- isEmpty(): Check if empty

Time Complexity: All operations are $O(1)$ Space Complexity: $O(n)$ for n elements

Stack

Operations on empty stack

- push(10): Stack becomes [10]
- push(20): Stack becomes [10, 20] (20 at front)
- push(30): Stack becomes [10, 20, 30] (30 at front)
- pop(): Returns 30, stack becomes [10, 20]

Insight

- Only top element is accessible
- No random access!
- Perfect for: Backtracking, recursion simulation, undo operations

Stack: Array Implementation

Key Points

- Track front index
- `front = -1` means empty stack
- `push`: increment `front`, add element
- `pop`: return element, decrement `front`

Time: $O(1)$ for all operations

Queues

Queue: Main Idea

Core Concept

- Add elements at rear
- Remove elements from front
- Operations at TWO ends

Key Operations

- push(x): Add element at rear
- pop(): Remove and return element from front
- front(): Return element without removing
- isEmpty(): Check if empty

Time Complexity: All operations are $O(1)$ Space Complexity: $O(n)$ for n elements

Queue

Operations on empty queue

- push(10): Queue becomes [10]
- push(20): Queue becomes [10, 20]
- push(30): Queue becomes [10, 20, 30]
- pop(): Returns 10, queue becomes [20, 30]

Insight

- Elements processed in order they arrive
- Fair scheduling: first come, first served
- Perfect for: BFS, task scheduling, buffering

Queue: Array Implementation

Two Options

Simple Array

- Easy to implement
- Wasted space issue

Circular Array

- Solves wasted space
- Fixed max size

Track front and back indices Update properly on push and pop!

Queue: Simple Array Problem

Issue with Simple Array

- front and back keep moving forward
- Eventually reach array end
- But array has empty space at beginning!

Example: After operations, array has only 1 element at index 2

- But can't push because back is at array end
- Beginning of array (indices 0, 1) is wasted

Wasted space! Can't push even though array has room.

Queue: Circular Array

Solution

- Wrap around to beginning when reaching end
- Use modulo arithmetic: `index = (index + 1) % MAX_SIZE`

Example: Front at index 3, rear at index 0

- Elements at indices 3, 4, 0 are used
- Indices 1, 2 are free
- Next enqueue goes to index 1 (wraps around)

Implementation Nits:

- Empty: `front == -1`
- Full: `(back + 1) % MAX_SIZE == front`
- Push: `back = (back + 1) % MAX_SIZE`
- Pop: `front = (front + 1) % MAX_SIZE`

Practice Problems

Problem 1: Valid Parentheses

Problem Statement: Given string containing (,), {, }, [,], determine if valid (every opening bracket has matching closing bracket in correct order).

Intuition

- When we see closing bracket, we need to match with MOST RECENT opening

Solution

1. Push all opening brackets onto stack
2. For closing bracket: check if matches stack top
3. If match: pop from stack
4. If no match: invalid
5. At end: stack should be empty

Time: $O(n)$ | Space: $O(n)$

Soln:

Input: "{[()]}"

Trace through each character:

- {: Push → Stack: [{]
- [: Push → Stack: [{, []
- (: Push → Stack: [{, [, (]
-): Matches top (, pop → Stack: [{, []
-]: Matches top [, pop → Stack: [{]
- }: Matches top {, pop → Stack: []

Valid! Stack is empty.

Time: $O(n)$ - single pass Space: $O(n)$ - worst case all opening brackets

Problem 2: Next Greater Element

Problem Statement For each element in array, find next element to right that is greater than itself. If none exists, output -1.

Intuition

- Stack can help “pop” smaller elements
- Traverse from right to left

Solution

1. Traverse array from right to left
2. Use stack to keep potential next greater elements
3. For each element:
   - Pop from stack until top is greater or stack empty
   - If stack empty: next greater is -1
   - Else: next greater is stack top
   - Push current element onto stack!

Time: $O(n)$ | Space: $O(n)$

Problem 3: Implement Queue using Stacks

Problem Statement Implement a FIFO queue using only LIFO stacks.

Insight

- Stack reverses order
- Two stacks = reverse twice -> original order!

Solution

- `input` stack: for push operations
- `output` stack: for pop operations
- When pop needed:
  - If `output` empty: move all from `input` to `output`
  - Pop from `output`

Soln:

Operations: push(1), push(2), pop(), push(3)

- push(1): Push to input stack
- push(2): Push to input stack
- pop(): Output empty, move all from input to output, pop from output
  - Stack 1 moves to Stack 2: [1, 2] $\longrightarrow$ [2, 1]
  - Pop 1 from output
- push(3): Push to input stack (output still has 2)
- Each element moved twice: input $\longrightarrow$ output $\longrightarrow$ removed
- Total operations = $2n$ for n elements

Space: $O(n)$ | Amortized Time: $O(1)$ per operation

Problem 4: Remove K Digits

Problem Statement Given string representing a non-negative integer, remove k digits to make smallest possible number.

Intuition

- Want smallest number $\longrightarrow$ remove large digits from left
- But need to maintain relative order -> monotonic stack!

Insight

- Use stack to build result
- Remove digit from stack if:
  - Current digit is smaller than stack top
  - We still have removals left ($k > 0$)
- This ensures smaller digits come first (more significant)

Solution

Time: $O(n)$ | Space: $O(n)$

Input: num = "1432219", k = 3

Trace through each digit:

- 1: Push → Stack: [1], k=3
- 4: Push → Stack: [1,4], k=3
- 3: 4>3, pop 4; push 3 → Stack: [1,3], k=2
- 2: 3>2, pop 3; push 2 → Stack: [1,2], k=1
- 2: 2≤2, push → Stack: [1,2,2], k=1
- 1: 2>1, pop 2; k=0, push 1 → Stack: [1,2,1,9]

Result: "1219" Edge cases: Leading zeros, all digits removed

Takeaway Problems!

Largest Rectangle

Largest rectangular area in histogram. Classic monotonic stack problem!

Task Scheduler

Schedule tasks with cooldown period. Greedy with priority queue!

When to Use What?

Use Stack when:

- Need LIFO ordering (undo operations, backtracking)
- Need to reverse something
- Matching pairs (parentheses, HTML tags)

Use Queue when:

- Need FIFO ordering (fair scheduling)
- Processing in arrival order,
- Level-order traversal (BFS)

Both are fundamental!

- Many complex problems reduce to stack/queue operations

Questions?

Thank you!