Stack & Queue - Complete DSA Study Notes

Table of Contents

PART 1: STACK

Stack - Core Concepts

Definition

Stack: A linear data structure following **LIFO (Last In First Out)** principle.

- Last element inserted is first to be removed
- Operations occur at ONE end only (called TOP)

Real-World Analogy: Stack of plates - you add/remove from top only

Fundamental Operations

Operation	Description	Time Complexity
push(x)	Add element to top	O(1)
pop()	Remove top element	O(1)
peek()/top()	View top element without removal	O(1)

Operation	Description	Time Complexity
isEmpty()	Check if stack is empty	O(1)
size()	Get number of elements	O(1)

Implementation Types

1. Array-Based Stack

```
Pseudocode:
class Stack {
    arr[MAX_SIZE]
    top = -1

    push(x):
        if top >= MAX_SIZE-1:
            return "Stack Overflow"
        top = top + 1
        arr[top] = x

    pop():
        if top < 0:
            return "Stack Underflow"
        x = arr[top]
        top = top - 1
        return x

    peek():
        if top < 0:
            return "Stack Empty"
        return arr[top]
}
```

Pros: Fast access, cache-friendly
Cons: Fixed size, wastage if not fully used

2. Linked List-Based Stack

```
Pseudocode:
class Node {
    data
    next
}
```

```
class Stack {
    head = NULL

    push(x):
        newNode = createNode(x)
        newNode.next = head
        head = newNode

    pop():
        if head == NULL:
            return "Stack Empty"
        temp = head
        head = head.next
        return temp.data

    peek():
        if head == NULL:
            return "Stack Empty"
        return head.data
}
```

Pros: Dynamic size, no overflow

Cons: Extra memory for pointers, slower than array

Stack Properties

- **LIFO Order:** Last pushed = First popped
- **Single-ended:** All operations at top
- **No random access:** Cannot access middle elements directly
- **Underflow:** Pop from empty stack
- **Overflow:** Push to full stack (array implementation)

Stack Patterns & Problems

Pattern 1: Basic Stack Operations

When to Use: Simple push/pop based problems

Problems:

1. **Implement Stack using Array** (Custom)
2. **Implement Stack using Linked List** (Custom)
3. **Min Stack** (LC 155)
4. **Stack using Queues** (LC 225)

Example - Min Stack:

```
Problem: Design stack that supports push, pop, top, and retrieving min in O(1)

Approach:
Use two stacks:
- mainStack: stores all elements
- minStack: stores minimum at each level

push(x):
    mainStack.push(x)
    if minStack.isEmpty() OR x <= minStack.top():
        minStack.push(x)

pop():
    if mainStack.top() == minStack.top():
        minStack.pop()
    mainStack.pop()

getMin():
    return minStack.top()

Time: O(1) all operations
Space: O(n)
```

Pattern 2: Balanced Parentheses

Recognition: Check if brackets/parentheses are balanced

Core Idea: Opening brackets push to stack, closing brackets must match top

Problems:

1. **Valid Parentheses** (LC 20) - Easy ⭐
2. **Generate Parentheses** (LC 22) - Medium
3. **Longest Valid Parentheses** (LC 32) - Hard
4. **Minimum Add to Make Valid** (LC 921) - Medium
5. **Remove Invalid Parentheses** (LC 301) - Hard

Approach Template:

```
Pseudocode:
isValid(s):
    stack = []
    map = {')': '(', '}': '{', ']': '['}

    for char in s:
        if char in opening_brackets:
            stack.push(char)
        else:
            if stack.isEmpty():
                return false
            if stack.pop() != map[char]:
                return false

    return stack.isEmpty()

Time: O(n)
Space: O(n)
```

Key Insight:

- Every closing bracket must have matching opening bracket
- Stack ensures correct nesting order
- Final stack must be empty

Pattern 3: Monotonic Stack

Definition: Stack maintaining elements in monotonic (increasing/decreasing) order

Two Types:

1. **Monotonic Increasing:** Bottom to top increases
2. **Monotonic Decreasing:** Bottom to top decreases

When to Use:

- Find next/previous greater/smaller element
- Problems with “nearest”, “next”, “previous”
- Optimization from $O(n^2)$ to $O(n)$

Problems:

1. **Next Greater Element I** (LC 496) - Easy ⭐
2. **Next Greater Element II** (LC 503) - Medium ⭐
3. **Daily Temperatures** (LC 739) - Medium ⭐
4. **Largest Rectangle in Histogram** (LC 84) - Hard ⭐
5. **Trapping Rain Water** (LC 42) - Hard ⭐
6. **Sum of Subarray Minimums** (LC 907) - Medium
7. **Online Stock Span** (LC 901) - Medium
8. **132 Pattern** (LC 456) - Medium

Template - Next Greater Element:

```
Pseudocode (Right Side):
nextGreater(arr):
    n = arr.length
    result = array of size n, all -1
    stack = []

    // Traverse from RIGHT to LEFT
    for i from n-1 to 0:
        // Pop smaller elements
        while !stack.isEmpty() AND stack.top() <= arr[i]:
            stack.pop()

        // If stack not empty, top is next greater
        if !stack.isEmpty():
            result[i] = stack.top()

        // Push current element
        stack.push(arr[i])

    return result

Time: O(n) - each element pushed/popped once
Space: O(n)
```

Template - Next Smaller Element:

```
Pseudocode:
nextSmaller(arr):
    stack = []
    result = [-1] * n

    for i from n-1 to 0:
        while !stack.isEmpty() AND stack.top() >= arr[i]:
            stack.pop()
```

```
        if !stack.isEmpty():
            result[i] = stack.top()

        stack.push(arr[i])

    return result
```

Four Variants:

```
1. Next Greater (right side) - Use decreasing stack, traverse right to left
2. Previous Greater (left side) - Use decreasing stack, traverse left to right
3. Next Smaller (right side) - Use increasing stack, traverse right to left
4. Previous Smaller (left side) - Use increasing stack, traverse left to right
```

Key Decision Table:

Need	Stack Type	Traversal Direction	Condition
Next Greater	Decreasing	Right to Left	pop while top ≤ current
Next Smaller	Increasing	Right to Left	pop while top ≥ current
Previous Greater	Decreasing	Left to Right	pop while top ≤ current
Previous Smaller	Increasing	Left to Right	pop while top ≥ current

Example - Daily Temperatures:

```
Problem: Given temperatures, return days until warmer temperature

Input: [73, 74, 75, 71, 69, 72, 76, 73]
Output: [1, 1, 4, 2, 1, 1, 0, 0]

Approach:
- Use monotonic decreasing stack
- Store indices instead of values
- When warmer temp found, calculate distance

Pseudocode:
dailyTemperatures(T):
    n = T.length
    result = [0] * n
    stack = []  // stores indices

    for i from 0 to n-1:
```

```
        while !stack.isEmpty() AND T[i] > T[stack.top()]:
            prevIndex = stack.pop()
            result[prevIndex] = i - prevIndex

        stack.push(i)

    return result

Why store indices? To calculate distance (days)
Time: O(n)
Space: O(n)
```

Example - Largest Rectangle in Histogram:

```
Problem: Given histogram heights, find largest rectangle area

Input: heights = [2,1,5,6,2,3]
Output: 10 (rectangle with height 5,6)

Approach:
- Use monotonic increasing stack
- When smaller bar found, calculate areas with popped bars
- Each bar extends left until smaller bar found

Pseudocode:
largestRectangle(heights):
    stack = []
    maxArea = 0

    for i from 0 to n:
        h = heights[i] if i < n else 0

        while !stack.isEmpty() AND h < heights[stack.top()]:
            height = heights[stack.pop()]
            width = i if stack.isEmpty() else i - stack.top() - 1
            maxArea = max(maxArea, height * width)

        stack.push(i)

    return maxArea

Key Insight:
- Stack maintains increasing heights
- When decreasing bar found, previous bars can extend
- Width = current position - position after popped index
```

```
Time: O(n)
Space: O(n)
```

Pattern 4: Expression Evaluation

When to Use: Evaluate mathematical expressions, convert notations

Problems:

1. **Evaluate Reverse Polish Notation** (LC 150) - Medium
2. **Basic Calculator** (LC 224) - Hard
3. **Basic Calculator II** (LC 227) - Medium
4. **Basic Calculator III** (LC 770) - Hard

Three Notations:

```
Infix:    A + B * C
Prefix:   + A * B C  (Polish notation)
Postfix:  A B C * +  (Reverse Polish notation)
```

Postfix Evaluation:

```
Pseudocode:
evalRPN(tokens):
    stack = []

    for token in tokens:
        if token is number:
            stack.push(int(token))
        else:  // operator
            b = stack.pop()
            a = stack.pop()
            result = apply_operator(a, operator, b)
            stack.push(result)

    return stack.pop()

Example: ["2", "1", "+", "3", "*"]
Step 1: push 2
Step 2: push 1
Step 3: pop 1,2 → 2+1=3, push 3
Step 4: push 3
Step 5: pop 3,3 → 3*3=9, push 9
```

```
Result: 9

Time: O(n)
Space: O(n)
```

Infix to Postfix Conversion:

```
Algorithm:
1. Create empty stack for operators
2. For each character:
   - If operand: add to output
   - If '(': push to stack
   - If ')': pop until '(' and add to output
   - If operator:
     * Pop operators with >= precedence
     * Push current operator
3. Pop remaining operators

Operator Precedence:
- Highest: ^
- Medium: *, /
- Lowest: +, -
```

Pattern 5: String Manipulation

When to Use: Remove elements, decode strings, simplify paths

Problems:

1. **Remove All Adjacent Duplicates** (LC 1047) - Easy
2. **Remove K Digits** (LC 402) - Medium
3. **Decode String** (LC 394) - Medium
4. **Simplify Path** (LC 71) - Medium
5. **Asteroid Collision** (LC 735) - Medium

Example - Remove Adjacent Duplicates:

```
Problem: Remove all adjacent duplicate characters

Input: "abbaca"
Output: "ca"
Explanation: "bb" removed → "aaca" → "aa" removed → "ca"
```

```
Approach:
Use stack to track characters

Pseudocode:
removeDuplicates(s):
    stack = []

    for char in s:
        if !stack.isEmpty() AND stack.top() == char:
            stack.pop()
        else:
            stack.push(char)

    return join(stack)

Time: O(n)
Space: O(n)
```

Example - Decode String:

```
Problem: Decode encoded string
Input: "3[a2[c]]"
Output: "accaccacc"

Approach:
Use stack to handle nested brackets

Pseudocode:
decodeString(s):
    stack = []
    currentNum = 0
    currentStr = ""

    for char in s:
        if char is digit:
            currentNum = currentNum * 10 + int(char)
        elif char == '[':
            stack.push(currentStr)
            stack.push(currentNum)
            currentStr = ""
            currentNum = 0
        elif char == ']':
            num = stack.pop()
            prevStr = stack.pop()
            currentStr = prevStr + currentStr * num
        else:
            currentStr += char
```

```
    return currentStr

Time: O(n * maxK) where maxK is max repeat count
Space: O(n)
```

Pattern 6: Function Call Stack Simulation

When to Use: DFS, backtracking, tree/graph traversal

Problems:

1. **Binary Tree Inorder Traversal** (LC 94) - Easy
2. **Binary Tree Preorder Traversal** (LC 144) - Easy
3. **Binary Tree Postorder Traversal** (LC 145) - Easy
4. **Flatten Binary Tree to Linked List** (LC 114) - Medium

Iterative Tree Traversal:

```
// INORDER: Left → Root → Right
inorderTraversal(root):
    stack = []
    result = []
    current = root

    while current != NULL OR !stack.isEmpty():
        // Go to leftmost node
        while current != NULL:
            stack.push(current)
            current = current.left

        // Process node
        current = stack.pop()
        result.add(current.val)

        // Go to right subtree
        current = current.right

    return result
```

```
// PREORDER: Root → Left → Right
preorderTraversal(root):
    if root == NULL: return []
```

```
    stack = [root]
    result = []

    while !stack.isEmpty():
        node = stack.pop()
        result.add(node.val)

        // Push right first (so left is processed first)
        if node.right != NULL:
            stack.push(node.right)
        if node.left != NULL:
            stack.push(node.left)

    return result
```

PART 2: QUEUE

Queue - Core Concepts

Definition

Queue: A linear data structure following **FIFO (First In First Out)** principle.

- First element inserted is first to be removed
- Insertion at REAR, Deletion at FRONT

Real-World Analogy: Line at ticket counter - first person in line served first

Fundamental Operations

Operation	Description	Time Complexity
enqueue(x)	Add element to rear	O(1)
dequeue()	Remove element from front	O(1)

Operation	Description	Time Complexity
front()/peek()	View front element	O(1)
isEmpty()	Check if queue is empty	O(1)
size()	Get number of elements	O(1)

Implementation Types

1. Array-Based Queue (Linear)

```
Pseudocode:
class Queue {
    arr[MAX_SIZE]
    front = -1
    rear = -1

    enqueue(x):
        if rear >= MAX_SIZE-1:
            return "Queue Full"
        if front == -1:
            front = 0
        rear = rear + 1
        arr[rear] = x

    dequeue():
        if front == -1 OR front > rear:
            return "Queue Empty"
        x = arr[front]
        front = front + 1
        return x
}
```

Problem: Wasted space - front moves forward, space before front unused

2. Circular Queue

```
Pseudocode:
class CircularQueue {
    arr[MAX_SIZE]
    front = -1
    rear = -1
```

```
    enqueue(x):
        if (rear + 1) % MAX_SIZE == front:
            return "Queue Full"
        if front == -1:
            front = 0
        rear = (rear + 1) % MAX_SIZE
        arr[rear] = x

    dequeue():
        if front == -1:
            return "Queue Empty"
        x = arr[front]
        if front == rear:  // Last element
            front = rear = -1
        else:
            front = (front + 1) % MAX_SIZE
        return x
}
```

Advantage: Reuses space, no wastage

3. Linked List-Based Queue

```
Pseudocode:
class Node {
    data
    next
}

class Queue {
    front = NULL
    rear = NULL

    enqueue(x):
        newNode = createNode(x)
        if rear == NULL:
            front = rear = newNode
            return
        rear.next = newNode
        rear = newNode

    dequeue():
        if front == NULL:
            return "Queue Empty"
        temp = front
        front = front.next
```

```
        if front == NULL:
            rear = NULL
        return temp.data
}
```

Queue Types

---

1. Simple Queue (Linear Queue)

- Standard FIFO
- Single-ended insertion and deletion

2. Circular Queue

- Last position connected to first
- Efficient memory utilization
- Used in: CPU scheduling, memory management

3. Priority Queue

- Elements have priorities
- Higher priority elements dequeued first
- Implementation: Heap (binary heap)

Operations:

```
insert(x, priority)
extractMax() or extractMin()
```

Problems:

1. **Kth Largest Element** (LC 215)
2. **Top K Frequent Elements** (LC 347)
3. **Merge K Sorted Lists** (LC 23)
4. **Find Median from Data Stream** (LC 295)

4. Deque (Double-Ended Queue)

- Insertion/deletion at both ends
- Can work as stack or queue

Operations:

```
insertFront(x)
insertRear(x)
deleteFront()
deleteRear()
```

Queue Patterns & Problems

Pattern 1: Basic Queue Operations

Problems:

1. **Implement Queue using Stacks** (LC 232) - Easy ⭐
2. **Implement Stack using Queues** (LC 225) - Easy
3. **Design Circular Queue** (LC 622) - Medium

Example - Queue using Stacks:

```
Approach: Use two stacks

Method 1 (Costly enqueue):
enqueue(x):
    while !stack1.isEmpty():
        stack2.push(stack1.pop())
    stack1.push(x)
    while !stack2.isEmpty():
        stack1.push(stack2.pop())

dequeue():
    return stack1.pop()

Time: Enqueue O(n), Dequeue O(1)

Method 2 (Costly dequeue):
enqueue(x):
    stack1.push(x)
```

```
dequeue():
    if stack2.isEmpty():
        while !stack1.isEmpty():
            stack2.push(stack1.pop())
    return stack2.pop()

Time: Enqueue O(1), Dequeue O(n) worst, O(1) amortized
```

Pattern 2: BFS (Breadth-First Search)

When to Use:

- Level-order traversal
- Shortest path in unweighted graph
- Problems requiring "level-by-level" processing

Problems:

1. **Binary Tree Level Order Traversal** (LC 102) - Medium ⭐
2. **Binary Tree Zigzag Level Order** (LC 103) - Medium
3. **Binary Tree Right Side View** (LC 199) - Medium
4. **Rotting Oranges** (LC 994) - Medium ⭐
5. **Word Ladder** (LC 127) - Hard
6. **Number of Islands** (LC 200) - Medium
7. **Shortest Path in Binary Matrix** (LC 1091) - Medium

BFS Template:

```
Pseudocode:
BFS(start):
    queue = [start]
    visited = set([start])

    while !queue.isEmpty():
        node = queue.dequeue()
        process(node)

        for neighbor in node.neighbors:
            if neighbor not in visited:
                visited.add(neighbor)
                queue.enqueue(neighbor)
```

Level-Order Traversal:

```
levelOrder(root):
    if root == NULL: return []

    result = []
    queue = [root]

    while !queue.isEmpty():
        levelSize = queue.size()
        currentLevel = []

        for i from 0 to levelSize-1:
            node = queue.dequeue()
            currentLevel.add(node.val)

            if node.left != NULL:
                queue.enqueue(node.left)
            if node.right != NULL:
                queue.enqueue(node.right)

        result.add(currentLevel)

    return result

Time: O(n) where n = number of nodes
Space: O(w) where w = maximum width of tree
```

Example - Rotting Oranges:

```
Problem: Grid with fresh(1) and rotten(2) oranges
Every minute, rotten oranges rot adjacent fresh oranges
Find minimum minutes to rot all oranges

Approach: Multi-source BFS

orangesRotting(grid):
    queue = []
    fresh = 0

    // Add all rotten oranges to queue
    for i in grid:
        for j in grid[i]:
            if grid[i][j] == 2:
                queue.enqueue((i, j, 0))  // row, col, time
            elif grid[i][j] == 1:
                fresh++
```

```
    minutes = 0
    directions = [(0,1), (1,0), (0,-1), (-1,0)]

    while !queue.isEmpty():
        row, col, time = queue.dequeue()
        minutes = time

        for dr, dc in directions:
            newRow, newCol = row + dr, col + dc
            if isValid(newRow, newCol) AND grid[newRow][newCol] == 1:
                grid[newRow][newCol] = 2
                fresh--
                queue.enqueue((newRow, newCol, time + 1))

    return minutes if fresh == 0 else -1

Time: O(m*n)
Space: O(m*n)
```

Pattern 3: Sliding Window Maximum (Monotonic Queue)

When to Use:

- Find maximum/minimum in sliding window
- Deque maintains elements in monotonic order

Problems:

1. **Sliding Window Maximum** (LC 239) - Hard ⭐
2. **Shortest Subarray with Sum at Least K** (LC 862) - Hard
3. **Longest Continuous Subarray With Absolute Diff** (LC 1438) - Medium

Approach:

```
Problem: Find maximum in each window of size k

Input: nums = [1,3,-1,-3,5,3,6,7], k = 3
Output: [3,3,5,5,6,7]

Use deque storing indices (not values)
Maintain decreasing order (front has max)

maxSlidingWindow(nums, k):
```

```
    deque = []
    result = []

    for i from 0 to n-1:
        // Remove elements outside window
        while !deque.isEmpty() AND deque.front() < i - k + 1:
            deque.removeFront()

        // Remove smaller elements (maintain decreasing)
        while !deque.isEmpty() AND nums[deque.back()] < nums[i]:
            deque.removeBack()

        deque.addBack(i)

        // Add to result after first window
        if i >= k - 1:
            result.add(nums[deque.front()])

    return result

Time: O(n) - each element added/removed once
Space: O(k)
```

Pattern 4: Task Scheduling

When to Use:

- Process tasks in order
- CPU scheduling problems
- Task dependencies

Problems:

1. **Task Scheduler** (LC 621) - Medium ⭐
2. **Reorganize String** (LC 767) - Medium
3. **Rearrange String k Distance Apart** (LC 358) - Hard

Example - Task Scheduler:

```
Problem: Tasks with cooldown n between same task
Find minimum time to complete all tasks

Input: tasks = ["A","A","A","B","B","B"], n = 2
```

```
Output: 8
Explanation: A -> B -> idle -> A -> B -> idle -> A -> B

Approach:
Use priority queue + queue

1. Count frequency of each task
2. Use max heap (priority queue) for task selection
3. Use queue to track cooldown

leastInterval(tasks, n):
    freq = count_frequency(tasks)
    maxHeap = createMaxHeap(freq.values())
    time = 0

    while !maxHeap.isEmpty():
        temp = []

        // Process n+1 time units
        for i from 0 to n:
            if !maxHeap.isEmpty():
                count = maxHeap.extractMax()
                if count > 1:
                    temp.add(count - 1)
            time++

            if maxHeap.isEmpty() AND temp.isEmpty():
                break

        // Add back to heap
        for count in temp:
            maxHeap.insert(count)

    return time

Time: O(n)
Space: O(1) - at most 26 tasks
```

Interview Questions & Answers

Q1: Difference between Stack and Queue?

Answer:

Aspect	Stack	Queue
Order	LIFO (Last In First Out)	FIFO (First In First Out)
Operations	Push/Pop from top	Enqueue at rear, Dequeue from front
Access	One end (top)	Two ends (front and rear)
Use Cases	Function calls, undo/redo, DFS	BFS, scheduling, buffering
Example	Stack of plates	Line at counter

Q2: When to use Stack vs Queue?

Stack:

- Need to reverse order (LIFO)
- Backtracking problems
- Expression evaluation
- DFS traversal
- Undo/Redo operations
- Browser history

Queue:

- Need to maintain order (FIFO)
- BFS traversal
- Level-order processing
- Task scheduling
- Buffering
- Producer-consumer problems

Q3: Explain Monotonic Stack. When to use it?

Answer:
Stack that maintains elements in either increasing or decreasing order.

Types:

1. **Monotonic Increasing:** Elements increase from bottom to top
2. **Monotonic Decreasing:** Elements decrease from bottom to top

When to Use:

- Find next/previous greater/smaller element
- Problems with “nearest”, “next larger/smaller”
- Optimize $O(n^2)$ brute force to O(n)

Problems:

- Next Greater Element
- Largest Rectangle in Histogram
- Daily Temperatures
- Trapping Rain Water

Key Insight: Pop elements that violate monotonic property - those elements won’t be answer for future elements

Q4: How to implement Queue using Stacks?

Answer:
Two methods:

Method 1: Costly Enqueue

```
- Enqueue: Transfer all to stack2, push new, transfer back → O(n)
- Dequeue: Simply pop from stack1 → O(1)
```

Method 2: Costly Dequeue (Better)

```
- Enqueue: Push to stack1 → O(1)
- Dequeue: If stack2 empty, transfer all from stack1 → Amortized O(1)
```

Why Method 2 better?
Each element transferred at most once, giving amortized O(1) for all operations

Q5: What is Priority Queue? How is it different from normal queue?

Answer:
Priority Queue:

- Elements have associated priorities
- Element with highest priority dequeued first
- Order based on priority, not insertion time

Normal Queue:

- FIFO order
- Insertion time determines order

Implementation:

- Usually implemented using Heap (Binary Heap)
- Operations: O(log n) for insert and extractMax/Min

Use Cases:

- Dijkstra's algorithm
- Huffman coding
- Task scheduling with priorities
- Event-driven simulation

Q6: Explain Circular Queue. Why is it better than linear queue?

Answer:
Circular Queue:

- Last position connected to first position
- Forms a circle
- Uses modulo arithmetic: (index + 1) % size

Advantages over Linear Queue:

1. **Memory Efficiency:** Reuses freed space
2. **No Wastage:** In linear queue, space before front is wasted
3. **Better Utilization:** All positions can be used

When Full:

```
(rear + 1) % size == front
```

Use Cases:

- CPU scheduling (Round Robin)
- Memory management
- Traffic systems

Q7: How to detect and remove loop in linked list using stack?

Answer:

Better Approach: Use Floyd's Cycle Detection (no stack needed)

Using Stack (Less Efficient):

```
detectLoop():
    stack = []
    visited = set()
    current = head

    while current != NULL:
        if current in visited:
            return true  // Loop detected
        visited.add(current)
        current = current.next

    return false

Time: O(n)
Space: O(n)
```

Floyd's Algorithm (Better):

```
Uses slow and fast pointers
Time: O(n)
Space: O(1)
```

Q8: Implement stack that supports getMin() in O(1)?

Answer:

Use auxiliary stack to track minimum

```
C++ Implementation:
class MinStack {
    stack<int> mainStack;
    stack<int> minStack;

public:
    void push(int x) {
        mainStack.push(x);
        if (minStack.empty() || x <= minStack.top())
            minStack.push(x);
```

```
    }

    void pop() {
        if (mainStack.top() == minStack.top())
            minStack.pop();
        mainStack.pop();
    }

    int top() {
        return mainStack.top();
    }

    int getMin() {
        return minStack.top();
    }
};
```

Space Optimization:
Can use single stack with formula: `2*x - currentMin`

Stack & Queue Complete Study Guide

Q9: What are applications of Stack in real world?

Answer:

Function Call Stack:

- Every function call pushed to call stack
- Return pops from stack
- Recursion uses implicit stack

Expression Evaluation:

- Infix to postfix conversion
- Calculator implementations
- Compiler syntax parsing

Undo/Redo Operations:

- Text editors
- Photoshop operations
- Browser back button

Backtracking Algorithms:

- Maze solving
- N-Queens problem
- Sudoku solver

Memory Management:

- Stack memory allocation
- Garbage collection

Browser History:

- Back/Forward navigation
- Tab management

Q10: What are applications of Queue in real world?

Answer:

CPU Scheduling:

- Process scheduling (FCFS, Round Robin)
- Job scheduling in OS

I/O Buffers:

- Keyboard buffer
- Printer spooling

- Network data packets

BFS Algorithm:

- Shortest path finding
- Level order traversal
- Web crawlers

Asynchronous Data Transfer:

- Pipes in OS
- Message queues
- Event handling

Real-world Queues:

- Customer service lines
- Call center systems
- Ticket booking

Resource Sharing:

- Shared printers
- CPU time sharing
- Disk scheduling

Q11: Explain Deque and its advantages?

Answer:

Deque (Double-Ended Queue):

- Operations at both front and rear
- Can act as both stack and queue

Operations:

- `insertFront(x)`
- `insertRear(x)`
- `deleteFront()`
- `deleteRear()`
- `getFront()`
- `getRear()`

Advantages:

- **Flexibility**: Can be used as stack or queue
- **Efficient**: All operations O(1)
- **Sliding Window**: Ideal for sliding window maximum
- **Palindrome Checking**: Access from both ends

Use Cases:

- Sliding window problems
- Undo-Redo with limits
- Browser history with forward/back
- Job scheduling with priorities

Q12: How to reverse a queue using stack?

Answer:

Approach:

1. Dequeue all elements and push to stack
2. Pop all elements from stack and enqueue

```
reverseQueue(queue):
    stack = []

    // Dequeue all to stack
    while !queue.isEmpty():
```

```
        stack.push(queue.dequeue())

    // Push all from stack to queue
    while !stack.isEmpty():
        queue.enqueue(stack.pop())
```

Time: O(n)

Space: O(n)

Recursive Approach (No Stack):

```
reverseQueue(queue):
    if queue.isEmpty():
        return

    element = queue.dequeue()
    reverseQueue(queue)
    queue.enqueue(element)
```

Space: O(n) - recursion stack

Q13: Design a data structure that supports insert, delete, and getRandom in O(1)?

Answer:

Use HashMap + ArrayList/Vector

C++ Approach:

```
class RandomizedSet {
    vector<int> nums;           // Store elements
    unordered_map<int, int> pos; // Element -> index

public:
    bool insert(int val) {
        if (pos.find(val) != pos.end())
            return false;
```

```
        nums.push_back(val);
        pos[val] = nums.size() - 1;
        return true;
    }

    bool remove(int val) {
        if (pos.find(val) == pos.end())
            return false;

        // Move last element to position of removed element
        int lastElement = nums.back();
        int idx = pos[val];

        nums[idx] = lastElement;
        pos[lastElement] = idx;

        nums.pop_back();
        pos.erase(val);
        return true;
    }

    int getRandom() {
        return nums[rand() % nums.size()];
    }
};
```

Time: All operations O(1)

Space: O(n)

Q14: What is the difference between BFS and DFS? When to use each?

Answer:

Aspect	BFS	DFS
Data Structure	Queue	Stack (or Recursion)
Traversal	Level by level	Depth first
Path	Shortest path	Any path

Aspect	BFS	DFS
Space	O(w) width of tree	O(h) height of tree
Complete	Yes	No (may not terminate)

Use BFS when:

- Finding shortest path in unweighted graph
- Level-order traversal needed
- Solution likely near root
- Need to explore all neighbors first

Use DFS when:

- Need to explore all paths
- Solution likely far from root
- Memory limited (DFS uses less space on trees)
- Backtracking problems
- Topological sorting
- Cycle detection

Q15: How to implement LRU Cache?

Answer:

Use Doubly Linked List + HashMap

Concept:

- Most recently used at front
- Least recently used at rear
- HashMap for O(1) access
- DLL for O(1) add/remove

C++ Approach:

```
class LRUCache {
    struct Node {
        int key, value;
        Node *prev, *next;
    };

    int capacity;
    unordered_map<int, Node*> cache;
    Node *head, *tail;

    void addNode(Node* node) {
        // Add to front (most recent)
        node->next = head->next;
        node->prev = head;
        head->next->prev = node;
        head->next = node;
    }

    void removeNode(Node* node) {
        node->prev->next = node->next;
        node->next->prev = node->prev;
    }

    void moveToFront(Node* node) {
        removeNode(node);
        addNode(node);
    }

public:
    LRUCache(int capacity) {
        this->capacity = capacity;
        head = new Node();
        tail = new Node();
        head->next = tail;
        tail->prev = head;
    }

    int get(int key) {
        if (cache.find(key) == cache.end())
            return -1;

        Node* node = cache[key];
        moveToFront(node);
        return node->value;
    }

    void put(int key, int value) {
```

```
        if (cache.find(key) != cache.end()) {
            Node* node = cache[key];
            node->value = value;
            moveToFront(node);
            return;
        }

        Node* node = new Node();
        node->key = key;
        node->value = value;

        cache[key] = node;
        addNode(node);

        if (cache.size() > capacity) {
            Node* lru = tail->prev;
            removeNode(lru);
            cache.erase(lru->key);
            delete lru;
        }
    }
};
```

Time: O(1) for both get and put

Space: O(capacity)

Key Points to Remember

Stack Essentials

1. LIFO Property:

```
Push order: A, B, C, D
Pop order:  D, C, B, A
```

2. When to Use Stack:

- ✅ Reverse something
- ✅ Match pairs (parentheses)
- ✅ Need to backtrack
- ✅ DFS traversal

- ✅ Expression evaluation
- ✅ Next/Previous greater/smaller

3. Monotonic Stack Rules:

For Next Greater:

- Use decreasing stack
- Traverse right to left
- Pop while top ≤ current

For Next Smaller:

- Use increasing stack
- Traverse right to left
- Pop while top ≥ current

4. Stack Overflow/Underflow:

- **Overflow**: Push to full stack (array implementation)
- **Underflow**: Pop from empty stack

5. Time Complexity:

- Push: O(1)
- Pop: O(1)
- Peek: O(1)
- Search: O(n) - need to pop elements

Queue Essentials

1. FIFO Property:

```
Enqueue order: A, B, C, D
Dequeue order: A, B, C, D
```

2. When to Use Queue:

- ✅ Maintain order

- ✅ BFS traversal
- ✅ Level-order processing
- ✅ Scheduling problems
- ✅ Buffer implementation

3. Queue Types:

- **Simple Queue**: Basic FIFO
- **Circular Queue**: Reuses space
- **Priority Queue**: Based on priority
- **Deque**: Both ends operations

4. Circular Queue Formula:

```
Next position: (index + 1) % size
Full condition: (rear + 1) % size == front
Empty condition: front == -1
```

5. Time Complexity:

- Enqueue: O(1)
- Dequeue: O(1)
- Front/Rear: O(1)
- Search: O(n)

Common Patterns

---

1. Expression Evaluation:

- **Postfix**: Use one stack for operands
- **Infix to Postfix**: Use one stack for operators

2. Balanced Parentheses:

- Opening brackets: Push
- Closing brackets: Pop and match
- Final stack must be empty

3. Next Greater Element:

- Maintain decreasing stack
- When larger element found, pop and set answer
- Current element becomes new potential answer

4. BFS Template:

- Use queue for level-order
- Process level by level
- Mark visited to avoid cycles

5. Sliding Window Maximum:

- Use deque
- Maintain decreasing order
- Front has maximum
- Remove out-of-window elements

Problem Recognition

Use Stack when you see:

- “Balanced”, “Valid parentheses”
- “Next greater/smaller”
- “Histogram”, “Rectangle”
- “Evaluate expression”
- “Remove duplicates”
- “Decode string”
- “Simplify path”

Use Queue when you see:

- “Level order”, “Layer by layer”
- “Shortest path” (unweighted)
- “Rotting”, “Spreading”

- “First K elements”
- “Recent calls”
- “Moving average”

Use Deque when you see:

- “Sliding window maximum/minimum”
- “Operations at both ends”
- “Recent K elements” with removal

Use Priority Queue when you see:

- “Kth largest/smallest”
- “Top K elements”
- “Merge K sorted”
- “Median of stream”
- “Task scheduler with priorities”

Optimization Tricks

1. Stack Space Optimization:

- Instead of storing actual elements, store indices
- Helps calculate distances/spans
- Example: Daily Temperatures, Stock Span

2. Two Stacks Technique:

- One for main data
- One for auxiliary info (min, max)
- Example: Min Stack, Max Stack

3. Queue using Stacks:

- Two stacks: input and output

- Transfer only when output empty
- Amortized O(1) operations

4. Monotonic Structure:

- Maintain order (increasing/decreasing)
- Pop elements that can't be answer
- Reduces $O(n^2)$ to O(n)

5. Multi-Source BFS:

- Add all sources to queue initially
- Process level by level
- Track time/distance
- Example: Rotting Oranges, 01 Matrix

Problem List by Pattern

Stack Problems

Pattern: Basic Operations

1. Implement Stack using Array (Custom)
2. Implement Stack using Linked List (Custom)
3. Min Stack (LC 155) - Easy ⭐
4. Max Stack (LC 716) - Easy

Pattern: Balanced Parentheses

5. Valid Parentheses (LC 20) - Easy ⭐
6. Generate Parentheses (LC 22) - Medium
7. Longest Valid Parentheses (LC 32) - Hard
8. Minimum Add to Make Valid (LC 921) - Medium
9. Minimum Remove to Make Valid (LC 1249) - Medium
10. Remove Invalid Parentheses (LC 301) - Hard

Pattern: Monotonic Stack

11. Next Greater Element I (LC 496) - Easy ⭐
12. Next Greater Element II (LC 503) - Medium ⭐
13. Daily Temperatures (LC 739) - Medium ⭐
14. Largest Rectangle in Histogram (LC 84) - Hard ⭐
15. Trapping Rain Water (LC 42) - Hard ⭐
16. Sum of Subarray Minimums (LC 907) - Medium
17. Online Stock Span (LC 901) - Medium
18. 132 Pattern (LC 456) - Medium
19. Maximum Width Ramp (LC 962) - Medium
20. Next Greater Node In Linked List (LC 1019) - Medium

Pattern: Expression Evaluation

21. Evaluate Reverse Polish Notation (LC 150) - Medium ⭐
22. Basic Calculator (LC 224) - Hard
23. Basic Calculator II (LC 227) - Medium ⭐
24. Basic Calculator III (LC 770) - Hard
25. Different Ways to Add Parentheses (LC 241) - Medium

Pattern: String Manipulation

26. Remove All Adjacent Duplicates (LC 1047) - Easy ⭐
27. Remove K Digits (LC 402) - Medium ⭐
28. Decode String (LC 394) - Medium ⭐
29. Simplify Path (LC 71) - Medium
30. Asteroid Collision (LC 735) - Medium
31. Backspace String Compare (LC 844) - Easy

Pattern: Tree Traversal

32. Binary Tree Inorder Traversal (LC 94) - Easy
33. Binary Tree Preorder Traversal (LC 144) - Easy
34. Binary Tree Postorder Traversal (LC 145) - Easy
35. Flatten Binary Tree (LC 114) - Medium
36. Binary Search Tree Iterator (LC 173) - Medium

Queue Problems

Pattern: Basic Operations

37. Implement Queue using Stacks (LC 232) - Easy ⭐
38. Implement Stack using Queues (LC 225) - Easy
39. Design Circular Queue (LC 622) - Medium
40. Design Circular Deque (LC 641) - Medium

Pattern: BFS Traversal

41. Binary Tree Level Order (LC 102) - Medium ⭐
42. Binary Tree Zigzag Level Order (LC 103) - Medium
43. Binary Tree Right Side View (LC 199) - Medium
44. Binary Tree Level Order II (LC 107) - Medium
45. Average of Levels (LC 637) - Easy
46. N-ary Tree Level Order (LC 429) - Medium
47. Maximum Level Sum (LC 1161) - Medium

Pattern: Grid BFS

48. Number of Islands (LC 200) - Medium ⭐
49. Rotting Oranges (LC 994) - Medium ⭐
50. Walls and Gates (LC 286) - Medium
51. 01 Matrix (LC 542) - Medium
52. Shortest Path in Binary Matrix (LC 1091) - Medium
53. Open the Lock (LC 752) - Medium
54. Snakes and Ladders (LC 909) - Medium

Pattern: Word Transformation

55. Word Ladder (LC 127) - Hard ⭐
56. Word Ladder II (LC 126) - Hard
57. Minimum Genetic Mutation (LC 433) - Medium

Pattern: Sliding Window with Deque

58. Sliding Window Maximum (LC 239) - Hard ⭐
59. Shortest Subarray with Sum at Least K (LC 862) - Hard
60. Longest Continuous Subarray (LC 1438) - Medium
61. Jump Game VI (LC 1696) - Medium

Pattern: Priority Queue (Heap)

62. Kth Largest Element (LC 215) - Medium ⭐
63. Top K Frequent Elements (LC 347) - Medium ⭐
64. Kth Largest Element in Stream (LC 703) - Easy
65. Find Median from Data Stream (LC 295) - Hard
66. Merge K Sorted Lists (LC 23) - Hard
67. Ugly Number II (LC 264) - Medium
68. Super Ugly Number (LC 313) - Medium
69. K Closest Points to Origin (LC 973) - Medium
70. Reorganize String (LC 767) - Medium

Pattern: Task Scheduling

71. Task Scheduler (LC 621) - Medium ⭐
72. Rearrange String k Distance Apart (LC 358) - Hard
73. Design Hit Counter (LC 362) - Medium
74. Number of Recent Calls (LC 933) - Easy
75. Time Needed to Buy Tickets (LC 2073) - Easy

Tips & Tricks

Stack Tips

1. Index vs Value Storage:

Store indices when:

- Need to calculate distance/span
- Need original position
- Working with arrays

Store values when:

- Only value matters
- No position needed

2. Direction Matters:

- **Next Greater (right side)**: Traverse right to left

- **Previous Greater (left side)**: Traverse left to right

3. Parentheses Matching:

Use map/dictionary:

```
closing -> opening
')' -> '('
'}' -> '{'
']' -> '['
```

4. Expression Evaluation:

- **Postfix**: Single stack for operands
- **Infix**: Two stacks (operands + operators)
- **Prefix**: Process right to left

5. Monotonic Stack Decision:

- Need GREATER? Use DECREASING stack
- Need SMALLER? Use INCREASING stack

Queue Tips

1. Level-Order Pattern:

Process by levels:

1. Get current level size
2. Process that many elements
3. Add children for next level

2. Multi-Source BFS:

- Add ALL sources initially
- Process together
- Track time/level

3. Circular Queue:

- Always use modulo: `(index + 1) % size`
- Full when: `(rear + 1) % size == front`

4. Priority Queue (Heap):

In C++ STL:

```
Max Heap: priority_queue<int>
Min Heap: priority_queue<int, vector<int>, greater<int>>
```

5. Deque for Window:

- Front has max/min
- Remove out-of-range from front
- Remove smaller/larger from back

Common Mistakes to Avoid

Stack:

- ❌ Checking empty before pop/peek
- ❌ Wrong loop condition in monotonic stack
- ❌ Forgetting to push current element
- ❌ Not handling edge cases (empty stack)

Queue:

- ❌ Not checking empty before dequeue
- ❌ Wrong level size in level-order traversal
- ❌ Forgetting to mark visited in BFS
- ❌ Circular queue overflow condition wrong

Time Complexity Cheat Sheet

Operation	Stack	Queue	Priority Queue
Insert	O(1)	O(1)	O(log n)
Delete	O(1)	O(1)	O(log n)
Peek	O(1)	O(1)	O(1)
Search	O(n)	O(n)	O(n)
Get Min/Max	O(1)*	-	O(1)

*With auxiliary stack

Space Complexity Patterns

- **Stack/Queue**: O(n) worst case
- **Monotonic Stack**: O(n) worst case
- **BFS**: O(w) where w = max width
- **DFS**: O(h) where h = height
- **Priority Queue**: O(k) for top k problems

Study Note Information

- **Version**: 1.0
- **Last Updated**: December 2024
- **Topics Covered**: Stack (36 problems) | Queue (39 problems)
- **Total Problems**: 75+
- **Patterns**: 15+ Major Patterns

Master these patterns and you'll handle any Stack/Queue problem in interviews!