Topic Introduction

Today we're diving into a classic and powerful data structure: the **stack**.

A stack is a collection of elements that supports two main operations:

- **Push:** Add an item to the top.
- **Pop:** Remove the item from the top.

Stacks follow the **Last-In-First-Out** (LIFO) principle: the last element you add is the first one you take out. Think of a stack of plates at a buffet—you add to the top, and when you take one, you grab the top one.

Why Are Stacks Useful?

Stacks shine whenever you need to remember what came before, process things in reverse order, or "undo" operations. They're super common in parsing expressions, recursion, tracking state changes, and simulating back/forward actions—like in browser history!

Example (not from today's problems):
Evaluating a string of parentheses `(()())`. You can use a stack to check if parentheses are balanced: push for every '(', pop for every ')', and at the end, the stack should be empty.

Today's three problems all involve **designing stack-based data structures with extra powers**:

- **Min Stack:** Supports retrieving the minimum element in constant time.
- **Max Stack:** Like Min Stack, but for the maximum.
- **Browser History:** Simulates browser navigation with back and forward buttons using stacks.

Why group these? Each one challenges you to extend the stack to support more operations efficiently—a super common interview pattern!

Problem 1: Min Stack

Leetcode 155. Min Stack

Problem Statement (in my words):

Design a stack that, in addition to push, pop, and top operations, can return the minimum element in constant time.

- `push(x)`: Push element x onto stack.
- `pop()`: Remove the top element.
- `top()`: Get the top element.
- `getMin()`: Retrieve the smallest element in the stack.

All operations should operate in O(1) time.

Example

```
MinStack minStack = new MinStack();
minStack.push(-2);
minStack.push(0);
minStack.push(-3);
minStack.getMin();   // returns -3
minStack.pop();
minStack.top();      // returns 0
minStack.getMin();   // returns -2
```

Thought Process

If you use a normal stack, `push`, `pop`, and `top` are easy. But `getMin`?
You could scan the stack to find the minimum, but that's O(n) time—not what we want.

Try This Yourself

What would happen if you just tracked the minimum value in a variable? What happens when you pop the minimum? Try this sequence:

```
Push 5
Push 3
Push 7
Pop
GetMin
Pop
GetMin
```

Brute-force Approach

Maintain a normal stack. For `getMin()`, iterate through the stack to find the minimum.

- **Time Complexity:** `push`, `pop`, `top` are O(1), but `getMin` is O(n).

Optimal Approach: Auxiliary Stack

Here's the core trick:
Use a second stack to keep track of the minimum so far.

- When you push a new value, also push onto the min stack the smaller of the new value or the current min.
- When you pop, pop from both stacks.

Now, the top of the min stack is always the minimum!

Dry-run Example:

Let's push: 5, 3, 7

Main stack: [5, 3, 7]

Min stack: [5, 3, 3]

Pop:

Main stack: [5, 3]

Min stack: [5, 3]

`getMin()` is top of min stack: 3

Try this sequence by hand:

Push 2, Push 1, Pop, GetMin

Python Solution

```
class MinStack:
    def __init__(self):
        self.stack = []      # Normal stack
        self.min_stack = []  # Tracks current min

    def push(self, val: int) -> None:
        self.stack.append(val)
        # If min_stack is empty or val is new min, push it; else repeat last min
        if not self.min_stack:
            self.min_stack.append(val)
        else:
            self.min_stack.append(min(val, self.min_stack[-1]))

    def pop(self) -> None:
        self.stack.pop()
        self.min_stack.pop()

    def top(self) -> int:
        return self.stack[-1]

    def getMin(self) -> int:
        return self.min_stack[-1]
```

Time Complexity: All operations are O(1)

Space Complexity: O(n) extra for the min stack

What’s Happening Here?

- `self.stack` holds everything you push.
- `self.min_stack` mirrors `self.stack` but at each position holds the minimum up to that point.
- On `push`, we compute the new min and append.
- On `pop`, we pop from both.
- `getMin` just peeks at the top of `min_stack`.

Trace Example:

Operations: push(5), push(3), push(7), getMin(), pop(), getMin()

- After push(5): stack=[5], min_stack=[5]
- push(3): stack=[5,3], min_stack=[5,3]
- push(7): stack=[5,3,7], min_stack=[5,3,3]
- getMin() -> 3
- pop(): stack=[5,3], min_stack=[5,3]
- getMin() -> 3

Try it Yourself:

push(2), push(0), push(1), pop(), getMin()

Take a moment to solve this on your own before peeking at the code!

Did you know you could do this with just one stack by storing (value, current_min) tuples? Try implementing that!

Problem 2: Max Stack

Leetcode 716. Max Stack

Problem Statement (in my words):

Design a stack that, in addition to usual stack operations, supports retrieving and removing the maximum element efficiently.

- `push(x)`: Push element x onto the stack.
- `pop()`: Remove the top element and return it.
- `top()`: Get the top element.
- `peekMax()`: Get the largest element in the stack.
- `popMax()`: Remove and return the largest element in the stack. If more than one exists, remove the one closest to the top.

Why is this like Min Stack?

We want to track the maximum value, not minimum. But **popMax** is sneakily tricky: we must find and remove the top-most maximum, not just peek at it.

Brute-force Approach

Use a normal stack. For `peekMax`, scan the stack to find the max (O(n)). For `popMax`, find the max, remove the highest-index occurrence by reconstructing the stack.

- **Time Complexity:** O(n) for `peekMax` and `popMax`

Optimal Approach: Two Stacks

Let’s focus on a simple yet effective technique that extends the Min Stack approach.

Idea:
Use two stacks:

- `stack`: Main stack
- `max_stack`: At each position, stores the max up to that point

For `popMax`:

- Pop elements from the main stack (and max_stack) onto a buffer stack until the top is the max.
- Pop the max.
- Push everything from the buffer back.

Pseudocode

```
class MaxStack:
    stack = []
    max_stack = []

    push(x):
        stack.push(x)
        if max_stack is empty:
            max_stack.push(x)
        else:
            max_stack.push(max(x, max_stack.top()))

    pop():
        max_stack.pop()
        return stack.pop()

    top():
        return stack.top()

    peekMax():
        return max_stack.top()
    popMax():
        max_val = max_stack.top()
        buffer = []
        # Pop until you find max
        while stack.top() != max_val:
            buffer.push(stack.pop())
            max_stack.pop()
        # Pop the max
        stack.pop()
        max_stack.pop()
        # Push back the buffer
        while buffer not empty:
            this_val = buffer.pop()
            stack.push(this_val)
               max_stack.push(max(this_val, max_stack.top() if max_stack not empty else
this_val))
        return max_val
```

Time Complexity:

- `push`, `pop`, `top`, `peekMax`: O(1)
- `popMax`: O(n) (in the worst case, need to pop all the way down)

Example

Operations: push(5), push(1), push(5), top(), popMax(), top(), peekMax(), pop()

- push(5): stack=[5], max_stack=[5]
- push(1): stack=[5,1], max_stack=[5,5]
- push(5): stack=[5,1,5], max_stack=[5,5,5]
- top() -> 5
- popMax(): remove top-most 5 (at index 2)
    - buffer=[]
    - pop 5 (is max), done
    - push nothing back
- stack=[5,1], max_stack=[5,5]
- top() -> 1
- peekMax() -> 5
- pop() -> 1

Try this yourself:

push(2), push(3), push(1), popMax(), peekMax()

Problem 3: Design Browser History

Leetcode 1472. Design Browser History

Problem Statement (in my words):

Simulate a browser's back and forward buttons.

- You start at a homepage.
- `visit(url)`: Visit a new page; clears all forward history.
- `back(steps)`: Go back up to `steps` times (or until you can't). Returns current page.
- `forward(steps)`: Go forward up to `steps` times (or until you can't). Returns current page.

How is this stack-based?

You can use two stacks! One to track the "back" history, one for "forward".

Brute-force Approach

Use a list to keep all history and a pointer, but updating and slicing lists can be less efficient and more error-prone.

Optimal Approach: Two Stacks

- **Back Stack:** Holds pages you can go back to.
- **Forward Stack:** Holds pages you can go forward to.
- **Current:** The current page.

On `visit(url)`:

- Push current page onto back stack.
- Set current to new url.
- Clear forward stack.

On `back(steps)`:

- While steps > 0 and back stack not empty:
    - Push current onto forward stack.
    - Pop from back stack and set as current.
    - Decrement steps.
- Return current.

On `forward(steps)`:

- While steps > 0 and forward stack not empty:
    - Push current onto back stack.
    - Pop from forward stack and set as current.
    - Decrement steps.
- Return current.

Pseudocode

```
class BrowserHistory:
    back_stack = []
    forward_stack = []
    current

    constructor(homepage):
        current = homepage

    visit(url):
        back_stack.push(current)
        current = url
        forward_stack.clear()

    back(steps):
        while steps > 0 and back_stack not empty:
            forward_stack.push(current)
            current = back_stack.pop()
            steps -= 1
        return current

    forward(steps):
        while steps > 0 and forward_stack not empty:
            back_stack.push(current)
            current = forward_stack.pop()
            steps -= 1
        return current
```

Time Complexity: All operations are O(steps) (linear in the number of steps moved, usually small compared to total history size). Most actions are O(1).

Example

Initialize with 'leetcode.com':
visit('google.com'), visit('facebook.com'), visit('youtube.com'), back(1), back(1), forward(1), visit('linkedin.com'), forward(2), back(2), back(7)

Trace:

- Homepage: leetcode.com
- visit(google.com): back=[leetcode.com], current=google.com, forward=[]
- visit(facebook.com): back=[leetcode.com,google.com], current=facebook.com, forward=[]

- visit(youtube.com): back=[leetcode.com,google.com,facebook.com], current=youtube.com, forward=[]
- back(1): forward=[youtube.com], current=facebook.com, back=[leetcode.com,google.com]
- back(1): forward=[youtube.com,facebook.com], current=google.com, back=[leetcode.com]
- forward(1): back=[leetcode.com,google.com], current=facebook.com, forward=[youtube.com]
- visit(linkedin.com): back=[leetcode.com,google.com,facebook.com], current=linkedin.com, forward=[]
- forward(2): (no effect, forward stack empty), current=linkedin.com
- back(2): forward=[linkedin.com,facebook.com], current=google.com, back=[leetcode.com]
- back(7): forward=[linkedin.com,facebook.com,google.com], current=leetcode.com, back=[]

Try This Yourself:

Set homepage to 'a.com', visit 'b.com', visit 'c.com', back(1), visit 'd.com', back(2), forward(1)

Reflect: Could you implement this with just one list and a pointer? What would be harder or easier?

Summary and Next Steps

We tackled three stack-based data structure design problems:

- **Min Stack:** Tracking the minimum in O(1) time using an auxiliary stack.
- **Max Stack:** Tracking the maximum, but with the twist of efficiently removing the top-most maximum.
- **Browser History:** Simulating browser navigation using two stacks.

Key Patterns:

- **Auxiliary Stacks:** Keeping extra information at each step enables constant-time min/max.
- **Two Stack Simulation:** For undo/redo or back/forward, two stacks capture the state transitions neatly.
- **State Mirroring:** For every push/pop in your main structure, update your helper stack(s) to mirror the state you need.

Common Pitfalls:

- Forgetting to update both stacks on push/pop.
- Not clearing forward history after a new visit in browser history.
- Not handling duplicates properly in popMax.

Action List:

- Solve all three problems on your own—even the one with code provided.
- Try solving Max Stack with a different approach (e.g., using a doubly-linked list and a TreeMap for true O(log n) popMax).
- Implement Browser History with a single list and pointer, and compare the tradeoffs.
- Explore other problems involving stacks, such as "Implement Queue using Stacks" or "Evaluate Reverse Polish Notation."
- Pay close attention to edge cases—especially empty stacks or popping the last occurrence.
- If you get stuck, take a walk and come back. Practice builds intuition!

Keep stacking up your skills—these patterns show up everywhere in interviews. Happy coding!