Stack Your Skills: Mastering Stack Patterns in Coding Interviews

Welcome back to another PrepLetter! Today, we’re diving into one of the most fundamental—and interview-friendly—data structures: the **stack**. You’ve probably heard the term before, but today we’ll peel back the layers, get comfortable with it, and then tackle three LeetCode problems that come up *all the time* in coding interviews.

What is a Stack? Why Do We Care?

A **stack** is a simple data structure that operates on a “last in, first out” (LIFO) principle. Imagine a stack of plates: you can only add or remove the top plate. That’s it! You can't pull a plate from the middle without making a mess.

How does it work?

A stack supports two main operations:

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

Some stacks also let you look at the top item without removing it, called **peek**.

When do we use stacks?

Stacks shine when you need to keep track of things in order, but only care about the most recent item. This pops up in:

- **Parentheses matching** (like checking if brackets are valid),
- **Undo operations** in editors,
- **Evaluating expressions** (think calculators),
- **Backtracking algorithms** (like traversing a maze).

A Simple Example: Reversing a String

Suppose you want to reverse the string `"hello"`. With a stack, you’d:

- Push each character onto the stack:

`h → e → l → l → o`

- Pop characters off one by one to build the reversed string:

`o → l → l → e → h`

This LIFO behavior means what goes in last comes out first.

Python Illustration:

```
stack = []
for ch in "hello":
    stack.append(ch)     # Push each char

reversed_str = ""
while stack:
    reversed_str += stack.pop()   # Pop from stack
```

```
# reversed_str is now "olleh"
```

Takeaway:
Whenever you need to process items in reverse order of their arrival, or you need to keep track of the most recent item, or need to match pairs (like open and close brackets), think stack!

Now that we've warmed up, let's tackle some problems that will really put our stack skills to the test. We'll start with a classic: **Vali d Parentheses**.

Problem 1: Valid Parentheses

Valid Parentheses – LeetCode #20

Problem Statement

Given a string containing just the characters `'('`, `')'`, `'{'`, `'}'`, `'['`, and `']'`, determine if the input string is valid. The string is valid if:

- Open brackets are closed by the same type of brackets.
- Open brackets are closed in the correct order.

Restated:
Does every opening bracket have a matching closing bracket, and are the brackets correctly nested?

Example Walkthrough

Input: `s = "({[]})"`
Output: `True`

- `(` opens, `{` opens, `[` opens, `]` closes `[`, `}` closes `{`, `)` closes `(`. All pairs match and close in order.

Let's try a trickier one:

Input: `s = "([)]"`
Output: `False`

- `(` opens, `[` opens, `)` tries to close `[` — mismatch! Invalid.

Try this on pen and paper:
`s = "{[()()]}"`
Does it return `True` or `False`?

Take a moment to try solving this on your own before reading the solution.

Stack-Based Solution

We'll use a stack to track open brackets. When we see an opening bracket, we push it. When we see a closing bracket, we check if it matches the last opening bracket (top of the stack). If not, it's invalid!

```
def isValid(s):
    stack = []
    mapping = {')': '(', '}': '{', ']': '['}  # maps closing to opening

    for char in s:
        if char in mapping.values():  # opening bracket
            stack.append(char)
        elif char in mapping:         # closing bracket
            if not stack or stack[-1] != mapping[char]:
                return False
            stack.pop()
        else:
            # Ignore any non-bracket characters (not needed for this problem)
            continue

    return not stack  # Stack should be empty if all brackets matched
```

Time and Space Complexity

- **Time:** O(n) — We process each character once.
- **Space:** O(n) — In the worst case, all are opening brackets.

Explanation

- The `stack` keeps track of unmatched opening brackets.
- The `mapping` dictionary lets us check if the current closing bracket matches the last opened one.
- If all brackets are matched and closed in order, the stack will be empty.

Let's walk through `"({[]})"`:

- Push `(`, stack = [`(`]
- Push `{`, stack = [`(`, `{`]
- Push `[`, stack = [`(`, `{`, `[`]
- See `]`: matches `[` (top), pop. Stack = [`(`, `{`]
- See `}`: matches `{` (top), pop. Stack = [`(`]
- See `)`: matches `(` (top), pop. Stack = []

Stack is empty, so it's valid!

Try this test case yourself:

`s = "((()))[]{}"`

Did you know this could also be solved using a counter for only one type of bracket? Try that after reading!

Now that you've seen a classic bracket-matching problem, let's see how stacks help with a slightly different string manipulation challenge.

Problem 2: Remove All Adjacent Duplicates In String

Remove All Adjacent Duplicates In String – LeetCode #1047

Problem Statement

You're given a string `s`. Remove all adjacent duplicate characters in the string until none remain. Return the final string.

Restated:
Scan from left to right. If two same characters are next to each other, remove them both. Repeat until no more adjacent duplicates.

Important:
One can think that only the last character matters, but you need to check all characters because removing one can create new adjacent duplicates! So, you need to keep checking!

Example Walkthrough

Input: `s = "abbaca"`
Output: `"ca"`

- `a` (push: stack = [`a`])
- `b` (push: [`a`, `b`])
- `b` (duplicate! pop: [`a`])
- `a` (duplicate! pop: [])
- `c` (push: [`c`])
- `a` (push: [`c`, `a`])

Stack: [`c`, `a`] → Final string is `"ca"`.

Try this one:
`s = "azxxzy"`
What's the result?

Take a moment to try solving this on your own before reading the solution.

Solution Using Stack

We'll use a stack to keep track of the resulting characters. If the current character matches the top of the stack, pop it (remove the duplicate). Otherwise, push it.

```
def removeDuplicates(s):
    stack = [] # In python, we can use a list as a stack
    for char in s:
        if stack and stack[-1] == char: # Check if top of stack matches current char
            stack.pop()   # Remove the duplicate
        else:
            stack.append(char)
    return ''.join(stack)
```

Time and Space Complexity

- **Time:** O(n) — Each character is pushed or popped at most once.
- **Space:** O(n) — The stack can grow up to the length of the string.

Explanation

- The stack simulates the process of removing adjacent duplicates.
- Each time you see a duplicate (i.e., the current character equals the top of the stack), you remove both.
- At the end, joining the stack gives the final result.

Let's compare to previous problem:
Instead of matching pairs of brackets, here you're matching identical neighbors. Same "push if not matching, pop if matching" logic, but now based on character equality.

Try this case on your own:
`s = "aabccba"`
What's the result after all adjacent duplicates are removed?

As you might guess, this could also be solved recursively, but the stack solution is more efficient and easier to understand for larger strings. Give recursion a shot if you're feeling adventurous!

Ready for a twist? Let's see how a stack helps when deletion rules become more "interactive".

Problem 3: Backspace String Compare

[Backspace String Compare – LeetCode #844]

Problem Statement

Given two strings `s` and `t`, where `#` means a backspace (delete the previous character), return `True` if they are equal after processing all backspaces.

Restated:

Type out `s` and `t`, where `#` erases the previous character (if there is one). Are the final strings equal?

Example Walkthrough

Input:
`s = "ab#c"`, `t = "ad#c"`
Output: `True`

Process `s`:

- `a` (push), `b` (push), `#` (pop `b`), `c` (push) → `"ac"`

Process `t`:

- `a` (push), `d` (push), `#` (pop `d`), `c` (push) → `"ac"`

Both result in `"ac"`.

Try this:
`s = "a##c"`, `t = "#a#c"`

Take a moment to try solving this on your own before reading the solution.

Stack-Based Solution

We simulate typing each string with a stack. Every character goes on the stack, unless it's a `#`, in which case we pop.

```
def build(string):
    stack = []
    for char in string:
        if char == '#':
            if stack:
                stack.pop()   # Backspace: remove previous character
        else:
            stack.append(char)
    return ''.join(stack)

def backspaceCompare(s, t):
    return build(s) == build(t)
```

Time and Space Complexity

- **Time:** O(n + m) — Where n and m are lengths of `s` and `t`.
- **Space:** O(n + m) — Stack for each string.

Explanation

- The `build` helper function creates the “final” string after applying backspaces.
- The main function compares the two results.

How is this different from the previous problem?

- Here, deletion is triggered by a special character (`#`), not just by adjacent duplicates.
- We need to process the entire string, simulating typing and deletion.

Try this case:

```
s = "bxj##tw"
t = "bxo#j##tw"
```

What do you get after processing both?

Bonus Challenge:

There’s a way to solve this in O(1) space by traversing both strings backwards and skipping characters as needed. Give it a shot once you’re comfortable with the stack method!

Summary and Next Steps

Today, you explored three problems that all showcase the power of stacks for string processing:

- **Valid Parentheses:** Matching pairs and nested structures.
- **Remove All Adjacent Duplicates:** Removing elements based on previous context.
- **Backspace String Compare:** Simulating deletion and undo actions.

Key patterns you should recognize:

- Stacks help manage “most recent” context—whether it’s an open bracket, duplicate character, or an action to undo.
- The LIFO property solves problems where order and reversibility matter.
- Many string and parsing problems can be tackled by simulating a process with a stack—look for clues in the problem description!

Common mistakes to avoid:

- Forgetting to check if the stack is empty before popping.
- Not handling all types of characters (especially when only specific ones trigger stack behavior).
- Mixing up stack order (remember, last in is first out).

Action List for You

- **Solve all three problems yourself:** Even if you’ve seen the solutions, coding them out builds muscle memory.
- **Practice dry-running examples by hand:** This helps you “think like a stack” before jumping to code.
- **Explore other stack-based problems:** Try “Min Stack,” “Evaluate Reverse Polish Notation,” or “Daily Temperatures.”
- **Check out other approaches:** See how recursion, two-pointer, or in-place methods work for these problems!
- **Don’t worry if you find some problems tricky:** Every coder has struggled with these before. Keep practicing and you’ll get there!

Stacks are a simple but mighty tool in your coding interview toolkit. With a bit of practice, you’ll start spotting these patterns everywhere. Keep stacking up that knowledge, and see you in the next PrepLetter!