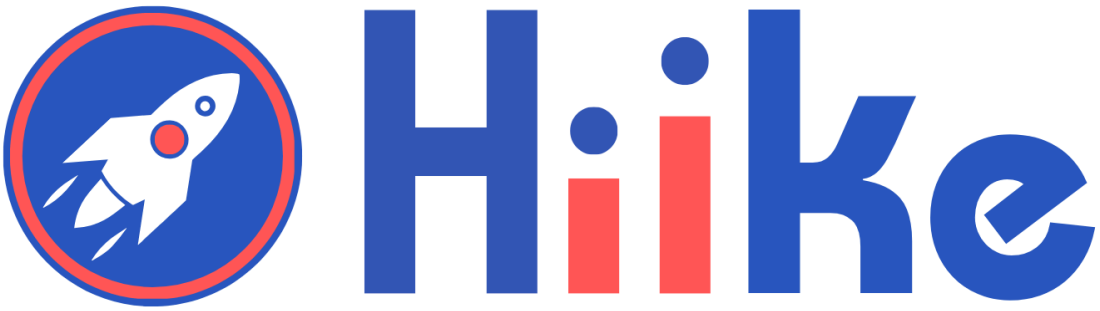
Stacks

Detailed Notes



A stack is a fundamental data structure that operates on a Last In, First Out (LIFO) principle. Here's a detailed look into the stack, its operations, and its application through example problems from LeetCode.

Characteristics of a Stack

LIFO Structure: The last element added to the stack is the first one to be removed.

Operations:

- Push: Add an element to the top.
- Pop: Remove the top element.
- Peek/Top: View the top element without removing it.
- IsEmpty: Check whether the stack is empty.

Real-World Analogies

- A stack of plates.
- Undo mechanism in software applications.

Implementation in Various Programming Languages

C++ (STL):

Use #include<stack>

- std::stack<int> stk; // declaring a stack
- stk.push(10); // pushing an item
- stk.pop(); // popping the top item
- stk.top(); // accessing the top item
- stk.empty(); // checking if the stack is empty

Java:

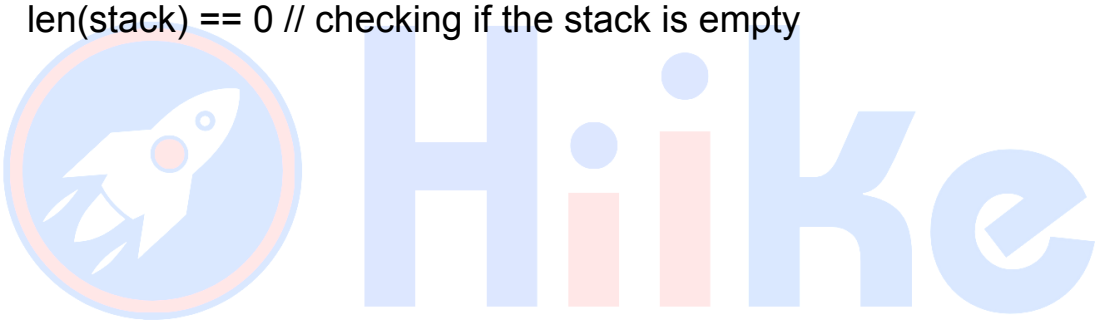
Use import java.util.Stack;

- Stack<Integer> stack = new Stack<>(); // declaring a stack
- stack.push(10); // pushing an item
- stack.pop(); // popping the top item
- stack.peek(); // accessing the top item
- stack.isEmpty(); // checking if the stack is empty

Python:

Lists in Python can be used as stacks.

- stack = []
- stack.append(10) // pushing an item
- stack.pop() // popping the top item
- stack[-1] // accessing the top item if stack is not empty
- len(stack) == 0 // checking if the stack is empty



LeetCode Problems with Pseudocode

Example 1: Valid Parentheses

Problem Statement: Check if the input string of parentheses is valid.

Pseudocode:

```
function isValid(s):
    Initialize an empty stack
    for each char in s:
        if char is an opening bracket:
            Push corresponding closing bracket onto stack
        else:
            if stack is empty or stack.pop() != char:
                return False
    return stack.isEmpty()
```

Solution Explanation:

The problem involves ensuring every opening bracket has a corresponding and correctly ordered closing bracket.

Why it works with a stack:

A stack allows you to process the last unclosed bracket first. When you encounter an opening bracket, you push the expected closing bracket onto the stack. For every closing bracket encountered, you check against the top of the stack (the last unmatched opening bracket). If they match, you pop from the stack, thus verifying that brackets close in the correct order. The stack should be empty at the end if all brackets are properly closed.

Example 2: Daily Temperatures

Problem Statement: Return a list that tells how many days you would have to wait until a warmer temperature.

Pseudocode:

```
function dailyTemperatures(T):
    Initialize result list with zeros
    Initialize an empty stack
    for each day i from 0 to length of T:
        while stack is not empty and T[i] > T[stack.top()]:
            index = stack.pop()
            result[index] = i - index
        Push i onto stack
    return result
```

Solution Explanation:

This problem requires you to look ahead to find when a certain condition (higher temperature) is met.

Why it works with a stack:

The stack is used to keep track of the indices of the days. You push the index of each day onto the stack until you find a day with a higher temperature. When a higher temperature is found, you resolve all previous colder days still in the stack by calculating the difference between the current day's index and the colder day's index. This uses the LIFO property to handle days in the reverse order they were seen until you find a warmer day.

Example 3: Largest Rectangle in Histogram

Problem Statement: Find the largest rectangle that can be formed in a histogram.

Pseudocode:

```
function largestRectangleArea(heights):
    Push a sentinel value (e.g., 0) onto heights
    Initialize an empty stack
    max_area = 0
    for each i from 0 to length of heights:
             while stack is not empty and current height < height at
stack.top():
            height = heights[stack.pop()]
            width = i if stack is empty else (i - stack.top() - 1)
            max_area = max(max_area, height * width)
        Push i onto stack
    return max_area
```

Solution Explanation:

This problem involves finding the maximum area rectangle in a histogram, which requires knowing the nearest smaller bar on both left and right for each bar.

Why it works with a stack:

As you traverse the histogram, you maintain a stack of bar indices where the heights are in increasing order. When you encounter a bar shorter than the bar at the stack's top, this indicates that the bar at the stack's top has found its right boundary. You can then pop the stack, calculate the area of the rectangle using the popped bar's height, and the current index as the right boundary and the new stack top as the left boundary. This ensures each bar is used efficiently to calculate the maximum possible rectangle it can form.

Example 4: Min Stack

Problem Statement: Design a stack that supports retrieving the minimum element in constant time.

Pseudocode:

```
class MinStack:
    Initialize stack and minStack

    function push(val):
        Push val onto stack
        if minStack is empty or val <= minStack.top():
            Push val onto minStack

    function pop():
        if stack.pop() == minStack.top():
            minStack.pop()

    function top():
        return stack.top()

    function getMin():
        return minStack.top()
```

Solution Explanation:

The challenge is to maintain a constant-time retrieval for the minimum element.

Why it works with a stack:

Alongside the main stack that stores all elements, a secondary stack (minStack) is used to keep track of the minimum elements. Each time an element is pushed onto the main stack, if it's smaller than or equal to the current minimum (the top element of minStack), it's also pushed onto minStack. When popping from the main stack, if the popped element is the same as the top of minStack, it's also popped from minStack. This way,

minStack always contains the minimum element of the elements currently in the main stack, allowing for constant-time retrieval.