Min Stack Solution (LeetCode 155)

Logic Explanation:

Min Stack Problem - LeetCode 155

Logic Explanation:

I used the standard library stack data structure and created two stacks:

1. 'st' - Main stack that stores all elements.
2. 'gt' - Auxiliary stack that keeps track of the minimum element at each step.

Push Operation:

- When pushing a value into 'st', check the top of 'gt':
  - If 'st' is empty, push the value in both 'st' and 'gt'.
  - If the new value is smaller than the top of 'gt', push it in 'gt'.
  - Otherwise, push the top element of 'gt' again to maintain the current minimum.

Pop Operation:

- Pop the top element from both 'st' and 'gt'.

Top Operation:

- Return 'st.top()'.

GetMin Operation:

- Return 'gt.top()' to get the minimum in O(1) time.

This ensures that all operations (push, pop, top, and getMin) run in O(1) time complexity.

Code Snippet:

```
// First solution for LeetCode problem 155
class MinStack {
public:
    stack<int> st;
```

```
    stack<int> gt;

    MinStack() {
        // Constructor (not needed in this method)
    }

    void push(int val) {
        if(st.size() == 0) {
            st.push(val);
            gt.push(val);
        } else {
            st.push(val);
            gt.push(min(val, gt.top()));
        }
    }

    void pop() {
        st.pop();
        gt.pop();
    }

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

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

Complexity Analysis:

Complexity Analysis:

- Push Operation: O(1) - Each push operation takes constant time.
- Pop Operation: O(1) - Removing top elements from both stacks takes constant time.
- Top Operation: O(1) - Accessing the top element of the stack takes constant time.
- GetMin Operation: O(1) - Fetching the minimum element from 'gt' takes constant time.

Overall, all operations run in **O(1) time complexity**, making this an optimal solution.

Test Cases and Execution:

Test Cases and Step-by-Step Execution:

Test Case 1:

Operations: ["MinStack","push","push","push","getMin","pop","top","getMin"]

Input: [[],[-2],[0],[-3],[],[],[],[]]

Expected Output: [null,null,null,null,-3,null,0,-2]

Step-by-Step Execution:

1. push(-2) -> st = [-2], gt = [-2] (min is -2)
2. push(0) -> st = [-2, 0], gt = [-2, -2] (min remains -2)
3. push(-3) -> st = [-2, 0, -3], gt = [-2, -2, -3] (new min is -3)
4. getMin() -> returns -3 (gt top)
5. pop()   -> removes -3 from both st and gt
6. top()   -> returns 0 (st top)
7. getMin() -> returns -2 (gt top)

Test Case 2:

Operations: ["MinStack","push","push","push","pop","getMin"]

Input: [[],[5],[1],[6],[],[]]

Expected Output: [null,null,null,null,null,1]

Step-by-Step Execution:

1. push(5) -> st = [5], gt = [5] (min is 5)
2. push(1) -> st = [5, 1], gt = [5, 1] (new min is 1)
3. push(6) -> st = [5, 1, 6], gt = [5, 1, 1] (min remains 1)
4. pop()   -> removes 6 from both st and gt

5. getMin() -> returns 1 (gt top)

Test Case 3:

Operations: ["MinStack","push","push","push","pop","pop","getMin"]

Input: [[],[10],[20],[5],[],[],[]]

Expected Output: [null,null,null,null,null,null,10]

Step-by-Step Execution:

1. push(10) -> st = [10], gt = [10] (min is 10)
2. push(20) -> st = [10, 20], gt = [10, 10] (min remains 10)
3. push(5) -> st = [10, 20, 5], gt = [10, 10, 5] (new min is 5)
4. pop()   -> removes 5 from both st and gt
5. pop()   -> removes 20 from both st and gt
6. getMin() -> returns 10 (gt top)