716. Max Stack

Nov. 25, 2017 | 57.5K views

Average Rating: 3.80 (40 votes)

Design a max stack that supports push, pop, top, peekMax and popMax.

1. push(x) -- Push element x onto stack.
2. pop() -- Remove the element on top of the stack and return it.
3. top() -- Get the element on the top.
4. peekMax() -- Retrieve the maximum element in the stack.
5. popMax() -- Retrieve the maximum element in the stack, and remove it. If you find more than one maximum elements, only remove the top-most one.

Example 1:

```
MaxStack stack = new MaxStack();
stack.push(5);
stack.push(1);
stack.push(5);
stack.top(); -> 5
stack.popMax(); -> 5
stack.top(); -> 1
stack.peekMax(); -> 5
stack.pop(); -> 1
stack.top(); -> 5
```

Note:

1. -1e7 <= x <= 1e7
2. Number of operations won't exceed 10000.
3. The last four operations won't be called when stack is empty.

Approach #1: Two Stacks [Accepted]

Intuition and Algorithm

A regular stack already supports the first 3 operations, so we focus on the last two.

For `peekMax`, we remember the largest value we've seen on the side. For example if we add `[2, 1, 5, 3, 9]`, we'll remember `[2, 2, 5, 5, 9]`. This works seamlessly with `pop` operations, and also it's easy to compute: it's just the maximum of the element we are adding and the previous maximum.

For `popMax`, we know what the current maximum (`peekMax`) is. We can pop until we find that maximum, then push the popped elements back on the stack.

Our implementation in Python will showcase extending the `list` class.

Java | Python | Copy

```python
class MaxStack(list):
    def push(self, x):
        m = max(x, self[-1][1] if self else None)
        self.append((x, m))

    def pop(self):
        return list.pop(self)[0]

    def top(self):
        return self[-1][0]

    def peekMax(self):
        return self[-1][1]

    def popMax(self):
        m = self[-1][1]
        b = []
        while self[-1][0] != m:
            b.append(self.pop())

        self.pop()
        map(self.push, reversed(b))
        return m
```

Complexity Analysis

- Time Complexity: $O(N)$ for the `popMax` operation, and $O(1)$ for the other operations, where N is the number of operations performed.
- Space Complexity: $O(N)$, the maximum size of the stack.

Approach #2: Double Linked List + TreeMap [Accepted]

Intuition

Using structures like Array or Stack will never let us `popMax` quickly. We turn our attention to tree and linked-list structures that have a lower time complexity for removal, with the aim of making `popMax` faster than $O(N)$ time complexity.

Say we have a double linked list as our "stack". This reduces the problem to finding which node to remove, since we can remove nodes in $O(1)$ time.

We can use a TreeMap mapping values to a list of nodes to answer this question. TreeMap can find the largest value, insert values, and delete values, all in $O(\log N)$ time.

Algorithm

Let's store the stack as a double linked list `dll`, and store a `map` from `value` to a `List` of `Node`.

- When we `MaxStack.push(x)`, we add a node to our `dll`, and add or update our entry `map.get(x).add(node)`.
- When we `MaxStack.pop()`, we find the value `val = dll.pop()`, and remove the node from our `map`, deleting the entry if it was the last one.
- When we `MaxStack.popMax()`, we use the `map` to find the relevant node to `unlink`, and return it's value.

The above operations are more clear given that we have a working `DoubleLinkedList` class. The implementation provided uses `head` and `tail` *sentinels* to simplify the relevant `DoubleLinkedList` operations.

A Python implementation was not included for this approach because there is no analog to *TreeMap* available.

Java | Copy

```java
class MaxStack {
    TreeMap<Integer, List<Node>> map;
    DoubleLinkedList dll;

    public MaxStack() {
        map = new TreeMap();
        dll = new DoubleLinkedList();
    }

    public void push(int x) {
        Node node = dll.add(x);
        if(!map.containsKey(x))
            map.put(x, new ArrayList<Node>());
        map.get(x).add(node);
    }

    public int pop() {
        int val = dll.pop();
        List<Node> L = map.get(val);
        L.remove(L.size() - 1);
        if (L.isEmpty()) map.remove(val);
        return val;
    }

    public int top() {
        return dll.peek();
    }
```

Complexity Analysis

- Time Complexity: $O(\log N)$ for all operations except `peek` which is $O(1)$, where N is the number of operations performed. Most operations involving `TreeMap` are $O(\log N)$.
- Space Complexity: $O(N)$, the size of the data structures used.

Analysis written by: @awice.

Rate this article:

Previous Next