<https://www.lintcode.com/problem/859/description>

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

* push(x) -- Push element x onto stack.
* pop() -- Remove the element on top of the stack and return it.
* top() -- Get the element on the top.
* peekMax() -- Retrieve the maximum element in the stack.
* 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.

**Attempt 1: 2023-05-31**

**The big difference than L155.Min Stack is L716.Max Stack requires a new method:**

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.

**Solution 1: Two Stacks (30 min)**

class MaxStack {

Stack<Integer> stack;

Stack<Integer> maxStack;

public MaxStack() {

stack = new Stack<Integer>();

maxStack = new Stack<Integer>();

}

public void push(int x) {

stack.push(x);

if(maxStack.isEmpty()) {

maxStack.push(x);

} else {

if(maxStack.peek() < x) {

maxStack.push(x);

} else {

maxStack.push(maxStack.peek());

}

}

}

// The elements from maxStack will removed without return

// but the elements from stack will removed with return

// The difference will help for popMax() method only push

// removed elements from stack to buffer

public int pop() {

maxStack.pop();

return stack.pop();

}

public int top() {

return stack.peek();

}

public int peekMax() {

return maxStack.peek();

}

public int popMax() {

int max = peekMax();

Stack<Integer> buffer = new Stack<Integer>();

while(top() != max) {

buffer.push(pop());

}

pop();

while(!buffer.isEmpty()) {

push(buffer.pop());

}

return max;

}

}

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.

**Refer to**

<https://aaronice.gitbook.io/lintcode/stack/max-stack>

### LeetCode Official - Approach #1: Two Stacks

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

ForpopMax, 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.

<https://leetcode.ca/2017-11-15-716-Max-Stack/>

The popMax() function. This is a highly complex function that requires careful handling. Since the maximum element may not always be at the top of the normal stack, it may be necessary to first pop some elements from the normal stack in order to locate it. The top element of the normal stack is considered the maximum only if it is identical to the top element of the maximum stack. Once the maximum element has been popped, any other elements that were previously removed from the normal stack must be re-added. To accomplish this, a new stack is used to store all the popped elements until the maximum element is identified. Once the maximum element has been popped and retrieved, each element in the new stack must be pushed back onto the normal stack, to ensure that both the normal stack and the maximum stack contain the correct values.

class MaxStack {

Stack<Integer> stack;

Stack<Integer> maxStack;

public MaxStack() {

stack = new Stack();

maxStack = new Stack();

}

public void push(int x) {

int max = maxStack.isEmpty() ? x : maxStack.peek();

maxStack.push(max > x ? max : x);

stack.push(x);

}

public int pop() {

maxStack.pop();

return stack.pop();

}

public int top() {

return stack.peek();

}

public int peekMax() {

return maxStack.peek();

}

public int popMax() {

int max = peekMax();

Stack<Integer> buffer = new Stack();

while (top() != max) buffer.push(pop());

pop();

while (!buffer.isEmpty()) push(buffer.pop());

return max;

}

}

**Example**

e.g

s.push(2);

s.push(9);

s.push(1);

s.push(5);

s.push(3);

s.popMax();

int result = s.peekMax();

stack maxStack stack maxStack buffer stack maxStack buffer

=== === === === === === === ===

3 9 b x

--- --- --- --- --- --- --- ---

5 9 popMax() b x 3 5

--- --- ======> --- --- --- ======> --- --- ---

1 9 b x 1 5 5 r

--- --- --- --- --- --- --- ---

9 9 x x 5 1 2 r

--- --- --- --- --- --- --- ---

2 2 2 2 3 2 2 r

=== === === === === === === ===

b means move to buffer r means return back to stack

x means permanently removed and maxStack based on push(x)

Note: especially maxStack elements

not move to buffer based on pop()

* Time Complexity: O(N) for the popMaxoperation, 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.

**Solution 2: Double Linked List + TreeMap (30 min)**

class MaxStack {

DoubleLinkedList list;

TreeMap<Integer, List<Node>> map;

public MaxStack() {

list = new DoubleLinkedList();

map = new TreeMap<Integer, List<Node>>();

}

public void push(int x) {

Node node = list.add(x);

if(!map.containsKey(x)) {

map.put(x, new ArrayList<Node>());

}

map.get(x).add(node);

}

public int pop() {

int val = list.pop();

List<Node> L = map.get(val);

L.remove(L.size() - 1);

if(L.isEmpty()) {

map.remove(val);

}

return val;

}

public int top() {

return list.peek();

}

// Use TreeMap natural ascending sorted attribute

public int peekMax() {

return map.lastKey();

}

public int popMax() {

int max = peekMax();

List<Node> L = map.get(max);

Node maxNode = L.remove(L.size() - 1);

list.unlink(maxNode);

if(L.isEmpty()) {

map.remove(max);

}

return max;

}

}

class DoubleLinkedList {

Node head;

Node tail;

public DoubleLinkedList() {

head = new Node(0);

tail = new Node(0);

head.next = tail;

tail.prev = head;

}

// Add new node as last element on current list

// head -> ... -> new node -> tail

// head <- ... <- new node <- tail

public Node add(int val) {

Node node = new Node(val);

node.next = tail;

node.prev = tail.prev;

tail.prev.next = node;

tail.prev = node;

return node;

}

public Node unlink(Node node) {

node.prev.next = node.next;

node.next.prev = node.prev;

return node;

}

public int pop() {

return unlink(tail.prev).val;

}

public int peek() {

return tail.prev.val;

}

}

class Node {

Node prev;

Node next;

int val;

public Node(int val) {

this.val = val;

}

}

Time Complexity: O(logN) for all operations except peek which is O(1), where N is the number of operations performed. Most operations involving TreeMap are O(logN).

Space Complexity: O(N), the size of the data structures used.

**Refer to**

<https://aaronice.gitbook.io/lintcode/stack/max-stack>

### LeetCode Official - Approach #2: Double Linked List + TreeMap

**Intuition**

Using structures like Array or Stack will never let uspopMaxquickly. We turn our attention to tree and linked-list structures that have a lower time complexity for removal, with the aim of makingpopMaxfaster 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(logN) time.

**Algorithm**

Let's store the stack as a double linked listdll, and store amapfromvalueto aListofNode.

* When weMaxStack.push(x), we add a node to ourdll, and add or update our entrymap.get(x).add(node).
* When weMaxStack.pop(), we find the valueval = dll.pop(), and remove the node from ourmap, deleting the entry if it was the last one.
* When weMaxStack.popMax(), we use themapto find the relevant node tounlink, and return it's value.

The above operations are more clear given that we have a workingDoubleLinkedListclass. The implementation provided usesheadandtail\_sentinels\_to simplify the relevantDoubleLinkedListoperations.

A Python implementation was not included for this approach because there is no analog to\_TreeMap\_available.

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();

}

public int peekMax() {

return map.lastKey();

}

public int popMax() {

int max = peekMax();

List<Node> L = map.get(max);

Node node = L.remove(L.size() - 1);

dll.unlink(node);

if (L.isEmpty()) map.remove(max);

return max;

}

}

class DoubleLinkedList {

Node head, tail;

public DoubleLinkedList() {

head = new Node(0);

tail = new Node(0);

head.next = tail;

tail.prev = head;

}

public Node add(int val) {

Node x = new Node(val);

x.next = tail;

x.prev = tail.prev;

tail.prev = tail.prev.next = x;

return x;

}

public int pop() {

return unlink(tail.prev).val;

}

public int peek() {

return tail.prev.val;

}

public Node unlink(Node node) {

node.prev.next = node.next;

node.next.prev = node.prev;

return node;

}

}

class Node {

int val;

Node prev, next;

public Node(int v) {val = v;}

}

**Complexity Analysis**

* Time Complexity: O(logN) for all operations exceptpeekwhich is O(1), where N is the number of operations performed. Most operations involvingTreeMapare O(logN).
* Space Complexity: O(N), the size of the data structures used.