# 题目

设计一个最大栈数据结构，既支持栈操作，又支持查找栈中最大元素。

实现 MaxStack 类：

MaxStack() 初始化栈对象

void push(int x) 将元素 x 压入栈中。

int pop() 移除栈顶元素并返回这个元素。

int top() 返回栈顶元素，无需移除。

int peekMax() 检索并返回栈中最大元素，无需移除。

int popMax() 检索并返回栈中最大元素，并将其移除。如果有多个最大元素，只要移除 最靠近栈顶的那个。

示例：

输入

["MaxStack", "push", "push", "push", "top", "popMax", "top", "peekMax", "pop", "top"]

[[], [5], [1], [5], [], [], [], [], [], []]

输出

[null, null, null, null, 5, 5, 1, 5, 1, 5]

解释

MaxStack stk = new MaxStack();

stk.push(5); // [5] - 5 既是栈顶元素，也是最大元素

stk.push(1); // [5, 1] - 栈顶元素是 1，最大元素是 5

stk.push(5); // [5, 1, 5] - 5 既是栈顶元素，也是最大元素

stk.top(); // 返回 5，[5, 1, 5] - 栈没有改变

stk.popMax(); // 返回 5，[5, 1] - 栈发生改变，栈顶元素不再是最大元素

stk.top(); // 返回 1，[5, 1] - 栈没有改变

stk.peekMax(); // 返回 5，[5, 1] - 栈没有改变

stk.pop(); // 返回 1，[5] - 此操作后，5 既是栈顶元素，也是最大元素

stk.top(); // 返回 5，[5] - 栈没有改变

提示：

-107 <= x <= 107

最多调用 104 次 push、pop、top、peekMax 和 popMax

调用 pop、top、peekMax 或 popMax 时，栈中 至少存在一个元素

进阶：

试着设计解决方案：调用 top 方法的时间复杂度为 O(1) ，调用其他方法的时间复杂度为 O(logn) 。

# 分析

## 方法一：双栈

### Java版本

一个普通的栈可以支持前三种操作push(x)，pop()和top()，所以我们需要考虑的仅为后两种操作peekMax()和popMax()。

对于peekMax()，我们可以另一个栈来存储每个位置到栈底的所有元素的最大值。例如，如果当前第一个栈中的元素为 [2, 1, 5, 3, 9]，那么第二个栈中的元素为 [2, 2, 5, 5, 9]。在push(x) 操作时，只需要将第二个栈的栈顶和 xx 的最大值入栈，而在pop()操作时，只需要将第二个栈进行出栈。

对于popMax()，由于我们知道当前栈中最大的元素值，因此可以直接将两个栈同时出栈，并存储第一个栈出栈的所有值。当某个时刻，第一个栈的出栈元素等于当前栈中最大的元素值时，就找到了最大的元素。此时我们将之前出第一个栈的所有元素重新入栈，并同步更新第二个栈，就完成了popMax()操作。

**代码：**

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;

}

}

**复杂度分析：**

时间复杂度：O(n)。由于前四个操作的时间复杂度都是O(1)，而popMax()操作在最坏情况下需要将栈中的所有元素全部出栈再入栈，时间复杂度为O(n)。因此总的时间复杂度为O(n)。

空间复杂度：O(n)。

### C++版本

**思路：**

1、一个数据栈，一个最大栈

2、注意要同步  
 同时弹，同时压

class MaxStack {

public:

/\*\* initialize your data structure here. \*/

stack<int> num\_stk;

stack<int> max\_stk; //最大栈，元素个数与数字栈同步

MaxStack()

{

}

void push(int x)

{

if (max\_stk.size()==0 || x > max\_stk.top() )

max\_stk.push(x);

else

max\_stk.push(max\_stk.top());

//当前最大的压进栈

num\_stk.push(x);

}

int pop()

{

max\_stk.pop(); //同步

int tmp = num\_stk.top();

num\_stk.pop();

return tmp;

}

int top()

{

return num\_stk.top();

}

int peekMax()

{

return max\_stk.top();

}

int popMax()

{

int cur\_max = max\_stk.top();

stack<int> tmp;

while(top() != cur\_max) //从数组栈里找

{

tmp.push(num\_stk.top());

pop(); //2个栈要同步，一起弹

}

pop(); //找到第一个，弹出

while(tmp.size())

{

push(tmp.top());

//再从缓冲区倒回去2个栈同步，一起压新数据

tmp.pop();

}

return cur\_max;

}

};

另一种写法：

class MaxStack {

public:

stack<int> elStack;

stack<int> maxStack;

/\*\* initialize your data structure here. \*/

MaxStack() {

}

void push(int x) {

elStack.push(x);

if (maxStack.empty() || x >= maxStack.top()) {

maxStack.push(x);

}

}

int pop() {

int num = elStack.top();

elStack.pop();

if (num == maxStack.top()) {

maxStack.pop();

}

return num;

}

int top() {

return elStack.top();

}

int peekMax() {

return maxStack.top();

}

int popMax() {

int maxNum = maxStack.top();

maxStack.pop();

stack<int> tmp;

while(elStack.top() != maxNum) {

tmp.push(elStack.top());

elStack.pop();

}

elStack.pop();

while(!tmp.empty()) {

push(tmp.top());

tmp.pop();

}

return maxNum;

}

};

## 方法二：双向链表 + 平衡树

分析

使用线性的数据结构（例如数组和栈）无法在较短的时间复杂度内完成popMax()操作，因此我们可以考虑使用双向链表 + 平衡树，其中双向链表用来表示栈，平衡树中的每一个节点存储一个键值对，其中“键”表示某个在栈中出现的值，“值”为一个列表，表示这个值在双向链表中出现的位置，存储方式为指针。平衡树的插入，删除和查找操作都是O(logn) 的，而通过平衡树定位到双向链表中的某个节点后，对该节点进行删除也是O(1) 的，因此所有操作的时间复杂度都不会超过O(logn)。

算法

我们使用双向链表存储栈，使用带键值对的平衡树（Java中的TreeMap）存储栈中出现的值以及这个值在双向链表中出现的位置。

push(x)操作：在双向链表的末尾添加一个节点，并且在平衡树上找到 xx，给它的列表中添加一个位置，指向新的节点。

pop()操作：在双向链表的末尾删除一个节点，它的值为val，随后在平衡树上找到val，删除它的列表中的最后一个位置。

top()操作：返回双向链表中最后一个节点的值。

peekMax()操作：返回平衡树上的最大值。

popMax()操作：在平衡树上找到最大值和它对应的列表，得到列表中的最后一个位置，并将它在双向链表中和平衡树上分别删除。

代码：

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;}

}

**复杂度分析：**

时间复杂度：O(logn)。top()操作的时间复杂度为O(1)，其余操作的时间复杂度为O(logn)，因此总的时间复杂度为O(logn)。

空间复杂度：O(n)。

官方已经提供了这种思路：平衡树中的每一个节点存储一个键值对，其中“键”表示某个在栈中出现的值，“值”为一个列表。

但官方只提供了Java的解法，这里提供C++的写法，并充分运用了List的各种函数,可以简洁优雅的完成要求。

class MaxStack {

public:

/\*\* initialize your data structure here. \*/

list<int> l;

map<int, vector<list<int>::iterator>> mp;

MaxStack() {

}

void push(int x) {

l.push\_front(x);

mp[x].push\_back(l.begin());

}

int pop() {

int key = l.front();

mp[key].pop\_back();

if(mp[key].empty()) mp.erase(key);

l.pop\_front();

return key;

}

int top() {

return l.front();

}

int peekMax() {

//rbegin() 返回一个逆序迭代器，它指向容器的最后一个元素

return mp.rbegin()->first;

}

int popMax() {

int key = mp.rbegin()->first;

auto it = mp[key].back();

mp[key].pop\_back();

if(mp[key].empty()) mp.erase(key);

l.erase(it);

return key;

}

};