# 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

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;

}

};

思路和心得：

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;

}

};

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;

}

};