# 3、堆栈、队列

**程序员面试题精选**(02)－设计包含min函数的栈

题目：定义栈的数据结构，要求添加一个min函数，能够得到栈的最小元素。要求函数min、push以及pop的时间复杂度都是O(1)。 分析：这是去年google的一道面试题。

我看到这道题目时，第一反应就是每次push一个新元素时，将栈里所有逆序元素排序。这样栈顶元素将是最小元素。但由于不能保证最后push进栈的元素最先出栈，这种思路设计的数据结构已经不是一个栈了。

在栈里添加一个成员变量存放最小元素（或最小元素的位置）。每次push一个新元素进栈的时候，如果该元素比当前的最小元素还要小，则更新最小元素。

乍一看这样思路挺好的。但仔细一想，该思路存在一个重要的问题：如果当前最小元素被pop出去，如何才能得到下一个最小元素？

因此仅仅只添加一个成员变量存放最小元素（或最小元素的位置）是不够的。我们需要一个辅助栈。每次push一个新元素的时候，同时将最小元素（或最小元素的位置。考虑到栈元素的类型可能是复杂的数据结构，用最小元素的位置将能减少空间消耗）push到辅助栈中；每次pop一个元素出栈的时候，同时pop辅助栈。

参考代码：

#include <deque>

#include <assert.h>

template <typename T> class CStackWithMin

{

public:

CStackWithMin(void) {}

virtual ~CStackWithMin(void) {}

T& top(void);

const T& top(void) const;

void push(const T& value);

void pop(void);

const T& min(void) const;

private:

T> m\_data; // the elements of stack

size\_t> m\_minIndex; // the indices of minimum elements

};

// get the last element of mutable stack

template <typename T> T& CStackWithMin<T>::top()

{

return m\_data.back();

}

// get the last element of non-mutable stack

template <typename T> const T& CStackWithMin<T>::top() const

{

return m\_data.back();

}

// insert an elment at the end of stack

template <typename T> void CStackWithMin<T>::push(const T& value)

{

// append the data into the end of m\_data

m\_data.push\_back(value);

// set the index of minimum elment in m\_data at the end of m\_minIndex

if(m\_minIndex.size() == 0)

m\_minIndex.push\_back(0);

else

{

if(value < m\_data[m\_minIndex.back()])

m\_minIndex.push\_back(m\_data.size() - 1);

else

m\_minIndex.push\_back(m\_minIndex.back());

}

}

// erease the element at the end of stack

template <typename T> void CStackWithMin<T>::pop()

{

// pop m\_data

m\_data.pop\_back();

// pop m\_minIndex

m\_minIndex.pop\_back();

}

// get the minimum element of stack

template <typename T> const T& CStackWithMin<T>::min() const

{

assert(m\_data.size() > 0);

assert(m\_minIndex.size() > 0);

return m\_data[m\_minIndex.back()];

}

举个例子演示上述代码的运行过程：

步骤 数据栈 辅助栈 最小值

1.push 3 3 0 3

2.push 4 3,4 0,0 3

3.push 2 3,4,2 0,0,2 2

4.push 1 3,4,2,1 0,0,2,3 1

5.pop 3,4,2 0,0,2 2

6.pop 3,4 0,0 3

7.push 0 3,4,0 0,0,2 0

讨论：如果思路正确，编写上述代码不是一件很难的事情。但如果能注意一些细节无疑能在面试中加分。比如我在上面的代码中做了如下的工作：

??????????用模板类实现。如果别人的元素类型只是int类型，模板将能给面试官带来好印象；

??????????两个版本的top函数。在很多类中，都需要提供const和非const版本的成员访问函数；

??????????min函数中assert。把代码写的尽量安全是每个软件公司对程序员的要求；

??????????添加一些注释。注释既能提高代码的可读性，又能增加代码量，何乐而不为？

总之，在面试时如果时间允许，尽量把代码写的漂亮一些。说不定代码中的几个小亮点就能让自己轻松拿到心仪的Offer。

**程序员面试题精选**(18)－用两个栈实现队列

题目：某队列的声明如下：

template<typename T> class CQueue

{

public:

CQueue() {}

~CQueue() {}

void appendTail(const T& node); // append a element to tail

void deleteHead(); // remove a element from head

private:

T> m\_stack1;

T> m\_stack2;

};

分析：从上面的类的声明中，我们发现在队列中有两个栈。因此这道题实质上是要求我们用两个栈来实现一个队列。相信大家对栈和队列的基本性质都非常了解了：栈是一种后入先出的数据容器，因此对队列进行的插入和删除操作都是在栈顶上进行；队列是一种先入先出的数据容器，我们总是把新元素插入到队列的尾部，而从队列的头部删除元素。

我们通过一个具体的例子来分析往该队列插入和删除元素的过程。首先插入一个元素a，不妨把先它插入到m\_stack1。这个时候m\_stack1中的元素有{a}，m\_stack2为空。再插入两个元素b和c，还是插入到m\_stack1中，此时m\_stack1中的元素有{a,b,c}，m\_stack2中仍然是空的。

这个时候我们试着从队列中删除一个元素。按照队列先入先出的规则，由于a比b、c先插入到队列中，这次被删除的元素应该是a。元素a存储在m\_stack1中，但并不在栈顶上，因此不能直接进行删除。注意到m\_stack2我们还一直没有使用过，现在是让m\_stack2起作用的时候了。如果我们把m\_stack1中的元素逐个pop出来并push进入m\_stack2，元素在m\_stack2中的顺序正好和原来在m\_stack1中的顺序相反。因此经过两次pop和push之后，m\_stack1为空，而m\_stack2中的元素是{c,b,a}。这个时候就可以pop出m\_stack2的栈顶a了。pop之后的m\_stack1为空，而m\_stack2的元素为{c,b}，其中b在栈顶。

这个时候如果我们还想继续删除应该怎么办呢？在剩下的两个元素中b和c，b比c先进入队列，因此b应该先删除。而此时b恰好又在栈顶上，因此可以直接pop出去。这次pop之后，m\_stack1中仍然为空，而m\_stack2为{c}。

从上面的分析我们可以总结出删除一个元素的步骤：当m\_stack2中不为空时，在m\_stack2中的栈顶元素是最先进入队列的元素，可以pop出去。如果m\_stack2为空时，我们把m\_stack1中的元素逐个pop出来并push进入m\_stack2。由于先进入队列的元素被压到m\_stack1的底端，经过pop和push之后就处于m\_stack2的顶端了，又可以直接pop出去。

接下来我们再插入一个元素d。我们是不是还可以把它push进m\_stack1？这样会不会有问题呢？我们说不会有问题。因为在删除元素的时候，如果m\_stack2中不为空，处于m\_stack2中的栈顶元素是最先进入队列的，可以直接pop；如果m\_stack2为空，我们把m\_stack1中的元素pop出来并push进入m\_stack2。由于m\_stack2中元素的顺序和m\_stack1相反，最先进入队列的元素还是处于m\_stack2的栈顶，仍然可以直接pop。不会出现任何矛盾。

我们用一个表来总结一下前面的例子执行的步骤：

操作 m\_stack1 m\_stack2

append a {a} {}

append b {a,b} {}

append c {a,b,c} {}

delete head {} {b,c}

delete head {} {c}

append d {d} {c}

delete head {d} {}

总结完push和pop对应的过程之后，我们可以开始动手写代码了。参考代码如下：

///////////////////////////////////////////////////////////////////////

// Append a element at the tail of the queue

///////////////////////////////////////////////////////////////////////

template<typename T> void CQueue<T>::appendTail(const T& element)

{

// push the new element into m\_stack1

m\_stack1.push(element);

}

///////////////////////////////////////////////////////////////////////

// Delete the head from the queue

///////////////////////////////////////////////////////////////////////

template<typename T> void CQueue<T>::deleteHead()

{

// if m\_stack2 is empty,

// and there are some elements in m\_stack1, push them in m\_stack2

if(m\_stack2.size() <= 0)

{

while(m\_stack1.size() > 0)

{

T& data = m\_stack1.top();

m\_stack1.pop();

m\_stack2.push(data);

}

}

// push the element into m\_stack2

assert(m\_stack2.size() > 0);

m\_stack2.pop();

}

扩展：这道题是用两个栈实现一个队列。反过来能不能用两个队列实现一个栈。如果可以，该如何实现？

**程序员面试题精选**100题(24)－栈的push、pop序列

题目：输入两个整数序列。其中一个序列表示栈的push顺序，判断另一个序列有没有可能是对应的pop顺序。为了简单起见，我们假设push序列的任意两个整数都是不相等的。

比如输入的push序列是1、2、3、4、5，那么4、5、3、2、1就有可能是一个pop系列。因为可以有如下的push和pop序列：push 1，push 2，push 3，push 4，pop，push 5，pop，pop，pop，pop，这样得到的pop序列就是4、5、3、2、1。但序列4、3、5、1、2就不可能是push序列1、2、3、4、5的pop序列。

分析：这到题除了考查对栈这一基本数据结构的理解，还能考查我们的分析能力。

这道题的一个很直观的想法就是建立一个辅助栈，每次push的时候就把一个整数push进入这个辅助栈，同样需要pop的时候就把该栈的栈顶整数pop出来。

我们以前面的序列4、5、3、2、1为例。第一个希望被pop出来的数字是4，因此4需要先push到栈里面。由于push的顺序已经由push序列确定了，也就是在把4 push进栈之前，数字1，2，3都需要push到栈里面。此时栈里的包含4个数字，分别是1，2，3，4，其中4位于栈顶。把4 pop出栈后，剩下三个数字1，2，3。接下来希望被pop的是5，由于仍然不是栈顶数字，我们接着在push序列中4以后的数字中寻找。找到数字5后再一次push进栈，这个时候5就是位于栈顶，可以被pop出来。接下来希望被pop的三个数字是3，2，1。每次操作前都位于栈顶，直接pop即可。

再来看序列4、3、5、1、2。pop数字4的情况和前面一样。把4 pop出来之后，3位于栈顶，直接pop。接下来希望pop的数字是5，由于5不是栈顶数字，我们到push序列中没有被push进栈的数字中去搜索该数字，幸运的时候能够找到5，于是把5 push进入栈。此时pop 5之后，栈内包含两个数字1、2，其中2位于栈顶。这个时候希望pop的数字是1，由于不是栈顶数字，我们需要到push序列中还没有被push进栈的数字中去搜索该数字。但此时push序列中所有数字都已被push进入栈，因此该序列不可能是一个pop序列。

也就是说，如果我们希望pop的数字正好是栈顶数字，直接pop出栈即可；如果希望pop的数字目前不在栈顶，我们就到push序列中还没有被push到栈里的数字中去搜索这个数字，并把在它之前的所有数字都push进栈。如果所有的数字都被push进栈仍然没有找到这个数字，表明该序列不可能是一个pop序列。

基于前面的分析，我们可以写出如下的参考代码：

#include <stack>

/////////////////////////////////////////////////////////////////////////////

// Given a push order of a stack, determine whether an array is possible to

// be its corresponding pop order

// Input: pPush - an array of integers, the push order

// pPop - an array of integers, the pop order

// nLength - the length of pPush and pPop

// Output: If pPop is possible to be the pop order of pPush, return true.

// Otherwise return false

/////////////////////////////////////////////////////////////////////////////

bool IsPossiblePopOrder(const int\* pPush, const int\* pPop, int nLength)

{

bool bPossible = false;

if(pPush && pPop && nLength > 0)

{

const int \*pNextPush = pPush;

const int \*pNextPop = pPop;

// ancillary stack

std::stack<int>stackData;

// check every integers in pPop

while(pNextPop - pPop < nLength)

{

// while the top of the ancillary stack is not the integer

// to be poped, try to push some integers into the stack

while(stackData.empty() || stackData.top() != \*pNextPop)

{

// pNextPush == NULL means all integers have been

// pushed into the stack, can't push any longer

if(!pNextPush)

break;

stackData.push(\*pNextPush);

// if there are integers left in pPush, move

// pNextPush forward, otherwise set it to be NULL

if(pNextPush - pPush < nLength - 1)

pNextPush ++;

else

pNextPush = NULL;

}

// After pushing, the top of stack is still not same as

// pPextPop, pPextPop is not in a pop sequence

// corresponding to pPush

if(stackData.top() != \*pNextPop)

break;

// Check the next integer in pPop

stackData.pop();

pNextPop ++;

}

// if all integers in pPop have been check successfully,

// pPop is a pop sequence corresponding to pPush

if(stackData.empty() && pNextPop - pPop == nLength)

bPossible = true;

}

return bPossible;

**程序员面试题精选100题(39)-颠倒栈[数据结构]**

题目：用递归颠倒一个栈。例如输入栈{1, 2, 3, 4, 5}，1在栈顶。颠倒之后的栈为{5, 4, 3, 2, 1}，5处在栈顶。

分析：乍一看到这道题目，第一反应是把栈里的所有元素逐一pop出来，放到一个数组里，然后在数组里颠倒所有元素，最后把数组中的所有元素逐一push进入栈。这时栈也就颠倒过来了。颠倒一个数组是一件很容易的事情。不过这种思路需要显示分配一个长度为O(n)的数组，而且也没有充分利用递归的特性。

我们再来考虑怎么递归。我们把栈{1, 2, 3, 4, 5}看成由两部分组成：栈顶元素1和剩下的部分{2, 3, 4, 5}。如果我们能把{2, 3, 4, 5}颠倒过来，变成{5, 4, 3, 2}，然后在把原来的栈顶元素1放到底部，那么就整个栈就颠倒过来了，变成{5, 4, 3, 2, 1}。

接下来我们需要考虑两件事情：一是如何把{2, 3, 4, 5}颠倒过来变成{5, 4, 3, 2}。我们只要把{2, 3, 4, 5}看成由两部分组成：栈顶元素2和剩下的部分{3, 4, 5}。我们只要把{3, 4, 5}先颠倒过来变成{5, 4, 3}，然后再把之前的栈顶元素2放到最底部，也就变成了{5, 4, 3, 2}。

至于怎么把{3, 4, 5}颠倒过来……很多读者可能都想到这就是递归。也就是每一次试图颠倒一个栈的时候，现在栈顶元素pop出来，再颠倒剩下的元素组成的栈，最后把之前的栈顶元素放到剩下元素组成的栈的底部。递归结束的条件是剩下的栈已经空了。这种思路的代码如下：

// Reverse a stack recursively in three steps:

// 1. Pop the top element

// 2. Reverse the remaining stack

// 3. Add the top element to the bottom of the remaining stack

template<typename T> void ReverseStack(std::stack<T>& stack)

{

    if(!stack.empty())

    {

        T top = stack.top();

        stack.pop();

        ReverseStack(stack);

        AddToStackBottom(stack, top);

    }

}

我们需要考虑的另外一件事情是如何把一个元素*e*放到一个栈的底部，也就是如何实现AddToStackBottom。这件事情不难，只需要把栈里原有的元素逐一pop出来。当栈为空的时候，push元素*e*进栈，此时它就位于栈的底部了。然后再把栈里原有的元素按照pop相反的顺序逐一push进栈。

注意到我们在push元素*e*之前，我们已经把栈里原有的所有元素都pop出来了，我们需要把它们保存起来，以便之后能把他们再push回去。我们当然可以开辟一个数组来做，但这没有必要。由于我们可以用递归来做这件事情，而递归本身就是一个栈结构。我们可以用递归的栈来保存这些元素。

基于如上分析，我们可以写出AddToStackBottom的代码：

// Add an element to the bottom of a stack:

template<typename T> void AddToStackBottom(std::stack<T>& stack, T t)

{

    if(stack.empty())

    {

        stack.push(t);

    }

    else

    {

        T top = stack.top();

        stack.pop();

        AddToStackBottom(stack, t);

        stack.push(top);

    }

}