# 题目

使用栈实现队列的下列操作：

push(x) -- 将一个元素放入队列的尾部。

pop() -- 从队列首部移除元素。

peek() -- 返回队列首部的元素。

empty() -- 返回队列是否为空。

**示例:**

MyQueue queue = new MyQueue();

queue.push(1);

queue.push(2);

queue.peek(); // 返回 1

queue.pop(); // 返回 1

queue.empty(); // 返回 false

**说明:**

你只能使用标准的栈操作--也就是只有push to top, peek/pop from top, size和is empty操作是合法的。

你所使用的语言也许不支持栈。你可以使用list或者deque（双端队列）来模拟一个栈，只要是标准的栈操作即可。

假设所有操作都是有效的（例如，一个空的队列不会调用pop或者peek操作）。

# 分析

## 方法一：临时栈

与用队列实现栈的思想一致，都是采用一个中间的数据结构转换（类似适配器）。





**代码：**



class MyQueue {

private:

std::stack<int> \_data;

public:

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

MyQueue() {

}

/\*\* Push element x to the back of queue. \*/

void push(int x) {

std::stack<int> tmpStack;

while(!\_data.empty())

{

tmpStack.push(\_data.top());

\_data.pop();

}

tmpStack.push(x); //这个位置与用队列实现栈的不同

while(!tmpStack.empty())

{

\_data.push(tmpStack.top());

tmpStack.pop();

}

}

/\*\* Removes the element from in front of queue and returns that element. \*/

int pop() {

//不能直接使用top，需要pop删除元素

int data = \_data.top();

\_data.pop();

return data;

}

/\*\* Get the front element. \*/

int peek() {

return \_data.top();

}

/\*\* Returns whether the queue is empty. \*/

bool empty() {

return \_data.empty();

}

};

/\*\*

\* Your MyQueue object will be instantiated and called as such:

\* MyQueue\* obj = new MyQueue();

\* obj->push(x);

\* int param\_2 = obj->pop();

\* int param\_3 = obj->peek();

\* bool param\_4 = obj->empty();

\*/

## **方法二：两个栈**









**代码：**



**思路：**

将一个栈当作输入栈，用于压入push传入的数据；另一个栈当作输出栈，用于pop和peek操作。

每次pop或peek时，若输出栈为空则将输入栈的全部数据依次弹出并压入输出栈，这样输出栈从栈顶往栈底的顺序就是队列从队首往队尾的顺序。

**代码：**

class MyQueue {

private:

stack<int> inStack, outStack;

void in2out() {

while (!inStack.empty()) {

outStack.push(inStack.top());

inStack.pop();

}

}

public:

MyQueue() {}

void push(int x) {

inStack.push(x);

}

int pop() {

if (outStack.empty()) {

in2out();

}

int x = outStack.top();

outStack.pop();

return x;

}

int peek() {

if (outStack.empty()) {

in2out();

}

return outStack.top();

}

bool empty() {

return inStack.empty() && outStack.empty();

}

};

**复杂度分析：**

时间复杂度：push和empty为O(1)，pop和peek 为均摊O(1)。对于每个元素，至多入栈和出栈各两次，故均摊复杂度为O(1)。

空间复杂度：O(n)。其中n是操作总数。对于有n次push操作的情况，队列中会有n个元素，故空间复杂度为O(n)。