题目地址(232. 用栈实现队列)

https://leetcode-cn.com/problems/implement-queue-using-stacks/

入选理由

1. 这题贼经典，考察次数很多。

题目描述

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）。

难度

- 简单

标签

- 栈

前置知识

- 栈
- 队列

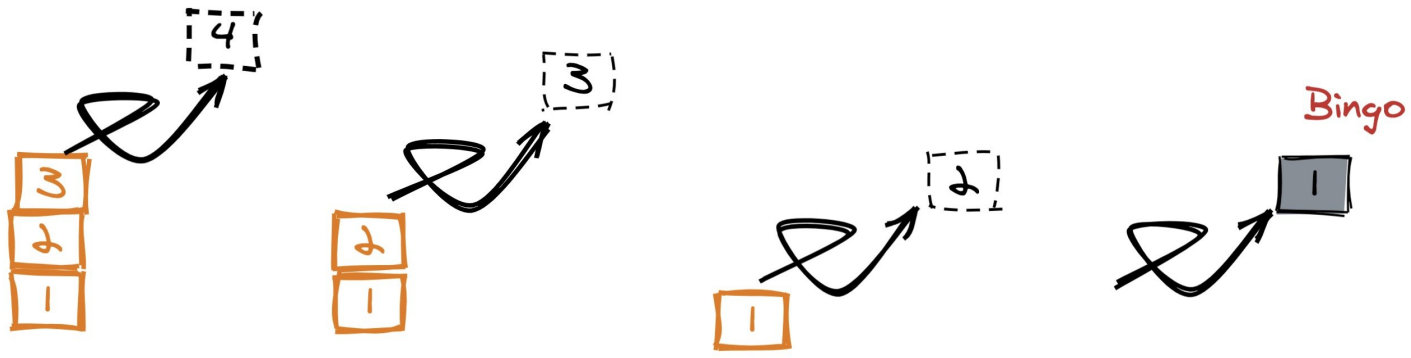
思路

题目要求用栈的原生操作来实现队列，也就是说需要用到 pop 和 push 但是我们知道 pop 和 push 都是在栈顶的操作，而队列的 enque 和 deque 则是在队列的两端的操作，这么一 看一个 stack 好像不太能完成。

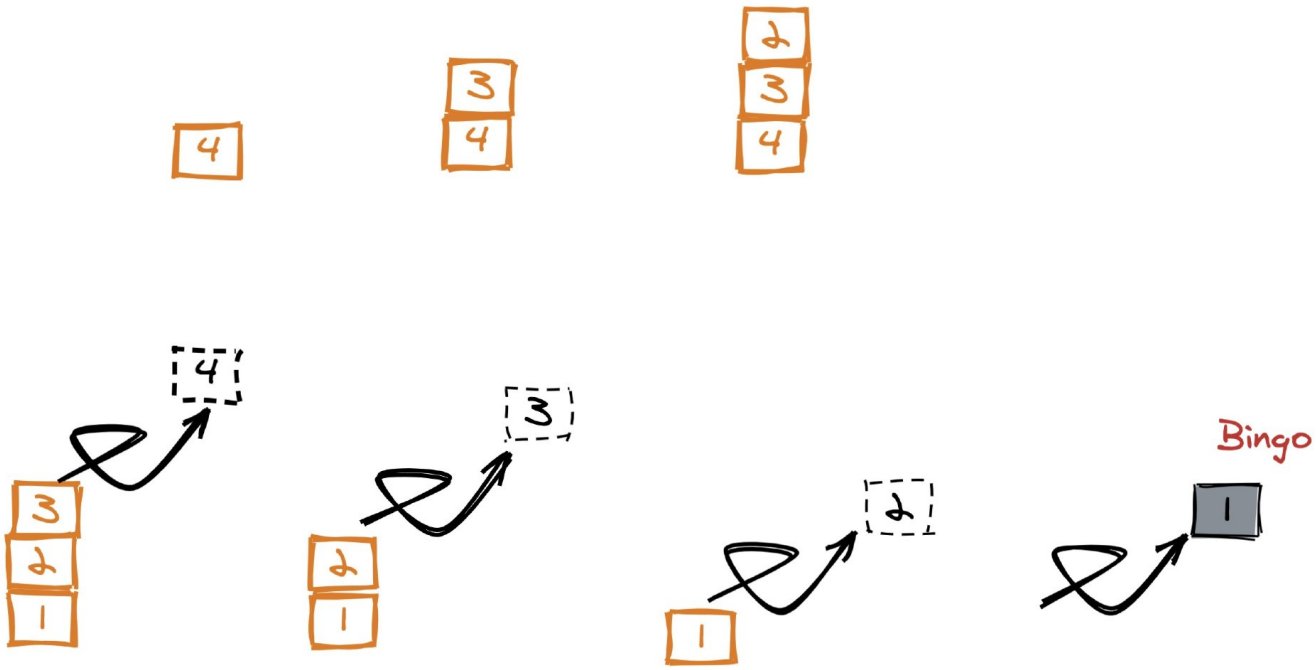
我们来分析一下过程。

假如向栈中分别 push 四个数字 `1, 2, 3, 4` ，那么此时栈的情况应该是：

如果此时按照题目要求 pop 或者 peek 的话， 应该是返回 1 才对，而 1 在栈底我们无法 直接操作。如果想要返回 1，我们首先要将 2，3，4 分别出栈才行。

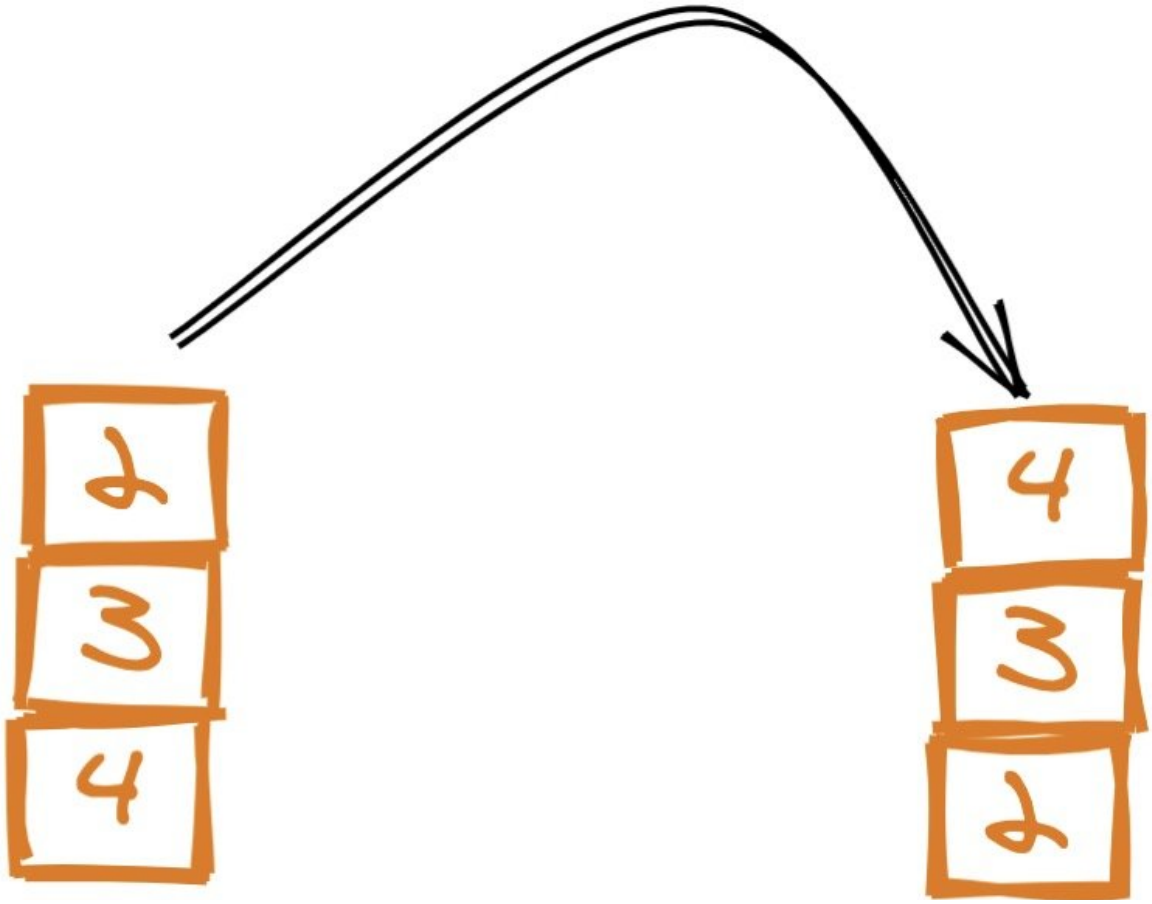


然而，如果我们这么做，1 虽然是正常返回了，但是 2，3，4 不就永远消失了么？ 一种简 答方法就是，将 2，3，4 **存** 起来。而题目又说了，只能使用栈这种数据结构，那么我 们考虑使用一个额外的栈来存放弹出的 2，3，4。

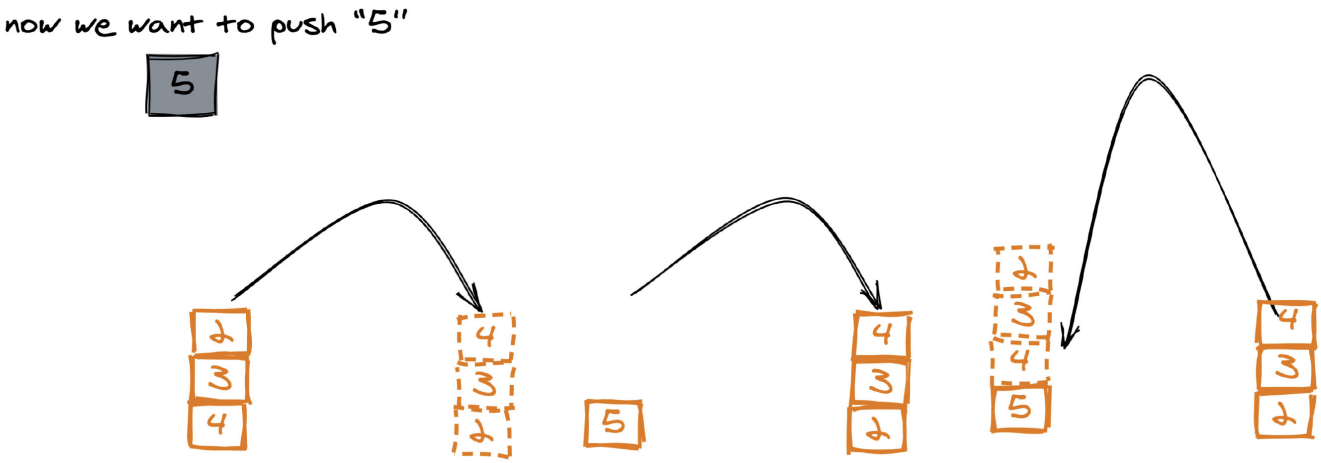


（pop 出 来不扔掉，而是存起来）

整个过程类似这样：



比如，这个时候，我们想 push 一个 5，那么大概就是这样的：



然而这一过程，我们也可以发生在 push 阶段。

总之，就是我们需要在 push 或者 pop 的时候，将数组在两个栈之间倒腾一次。

关键点

- 在 push 的时候利用辅助栈(双栈)

代码

部分代码参考自： [力扣官解]

其中 inStack 为写栈，outStack 为读栈。

- 语言支持：JS, Python, Java，CPP

Javascript Code:

```
var MyQueue = function () {
  this.inStack = [];
  this.outStack = [];
};

MyQueue.prototype.push = function (x) {
  this.inStack.push(x);
};
```

```
MyQueue.prototype.pop = function () {
  if (!this.outStack.length) {
    this.in2out();
  }
  return this.outStack.pop();
};

MyQueue.prototype.peek = function () {
  if (!this.outStack.length) {
    this.in2out();
  }
  return this.outStack[this.outStack.length - 1];
};

MyQueue.prototype.empty = function () {
  return this.outStack.length === 0 && this.inStack.length === 0;
};

MyQueue.prototype.in2out = function () {
  while (this.inStack.length) {
    this.outStack.push(this.inStack.pop());
  }
};
```

Python Code:

```
class MyQueue:

  def __init__(self):
      """
      Initialize your data structure here.
      """
      self.inStack = []
      self.outStack = []

  def push(self, x: int) -> None:
      """
      Push element x to the back of queue.
      """
      self.inStack.append(x)

  def pop(self) -> int:
      """
      Removes the element from in front of queue and returns that element.
      """
      if not self.outStack:
          while self.inStack:
              self.outStack.append(self.inStack.pop())

      return self.outStack.pop()
```

```
    def peek(self) -> int:
        """
        Get the front element.
        """
        if self.inStack:
            return self.inStack[0]
        else:
            return self.outStack[-1]

    def empty(self) -> bool:
        """
        Returns whether the queue is empty.
        """
        if not self.inStack and not self.outStack:
            return True
        else:
            return False
```

Java Code

```
class MyQueue {
    Deque<Integer> inStack;
    Deque<Integer> outStack;

    public MyQueue() {
        inStack = new ArrayDeque<Integer>();
        outStack = new ArrayDeque<Integer>();
    }

    public void push(int x) {
        inStack.push(x);
    }

    public int pop() {
        if (outStack.isEmpty()) {
            in2out();
        }
        return outStack.pop();
    }

    public int peek() {
        if (outStack.isEmpty()) {
            in2out();
        }
        return outStack.peek();
    }
```

```
    public boolean empty() {
        return inStack.isEmpty() && outStack.isEmpty();
    }

    private void in2out() {
        while (!inStack.isEmpty()) {
            outStack.push(inStack.pop());
        }
    }
}
```

CPP Code:

```
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： `O(N)` ，其中 N 为 栈中元素个数，因为每次我们都要倒腾一 次。
- 空间复杂度： `O(N)` ，其中 N 为 栈中元素个数，多使用了一个辅助栈，这 个辅助栈的大小和原栈的大小一样。

扩展

- 类似的题目有用队列实现栈，思路是完全一样的，大家有兴趣可以试一下。
- 栈混洗也是借助另外一个栈来完成的，从这点来看，两者有相似之处。

延伸阅读

实际上现实中也有使用两个栈来实现队列的情况，那么为什么我们要用两个 stack 来实现 一个 queue?

其实使用两个栈来替代一个队列的实现是为了在多进程中分开对同一个队列对读写操作。一 个栈是用来读的，另一个是用来写的。当且仅当读栈满时或者写栈为空时，读写操作才会发 生冲突。

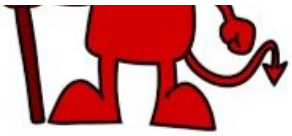
当只有一个线程对栈进行读写操作的时候，总有一个栈是空的。在多线程应用中，如果我们 只有一个队列，为了线程安全，我们在读或者写队列的时候都需要锁住整个队列。而在两个 栈的实现中，只要写入栈不为空，那么 `push` 操作的锁就不会影响到 `pop` 。

- reference
- further reading

更多题解可以访问我的 LeetCode 题解仓库：https://github.com/azl397985856/leetcode 。 目前已经 30K star 啦。

大家也可以关注我的公众号《力扣加加》获取更多更新鲜的 LeetCode 题解

欢迎长按关注



Originally posted by @azl397985856 in https://github.com/leetcode-pp/91alg-1/issues/21#issuecomment-639573715

上一页 下一页