232. 用栈实现队列

题目描述

请你仅使用两个栈实现**先入先出队列**。队列应当支持一般队列支持的所有操作（`push`、`pop`、`peek`、`empty`）：

实现 `MyQueue` 类：

- `void push(int x)`：将元素 x 推到队列的末尾。
- `int pop()`：从队列的开头移除并返回元素。
- `int peek()`：返回队列开头的元素。
- `boolean empty()`：如果队列为空，返回 `true`；否则，返回 `false`。

说明

1. 你**只能**使用标准的栈操作：
   - `push to top`、`peek/pop from top`、`size` 和 `is empty`。
2. 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。

示例

输入：

```
["MyQueue", "push", "push", "peek", "pop", "empty"]
[[], [1], [2], [], [], []]
```

输出：

```
[null, null, null, 1, 1, false]
```

解释：

```
MyQueue myQueue = new MyQueue();
myQueue.push(1); // queue is: [1]
myQueue.push(2); // queue is: [1, 2] (leftmost is front of the queue)
myQueue.peek(); // return 1
myQueue.pop(); // return 1, queue is [2]
myQueue.empty(); // return false
```

提示

- $1 \leq x \leq 9$
- 最多调用 100 次 `push`、`pop`、`peek` 和 `empty`。
- 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）。

进阶

你能否实现每个操作均摊时间复杂度为 $O(1)$ 的队列?

换句话说，执行 n 个操作的总时间复杂度为 $O(n)$，即使其中一个操作可能花费较长时间。

核心思想是用两个stack模拟queue

In [2]:
```python
class MyQueue:

    def __init__(self):
        self.stack_in = []
        self.stack_out = []

    def push(self, x: int) -> None:
        """
        有新元素进来，就往in里面push
        """
        self.stack_in.append(x)

    def pop(self) -> int:
        """
        获取队列的头部元素（第一个元素），并移除它。
        Removes the element from in front of queue and returns that element.
        """
        if self.empty():
            return None
        if self.stack_out:
            return self.stack_out.pop()
        else:
            # 这步是最关键的，就是相当于把 stack_in 按照反转的顺序依次放进stack_out中，然后pop最后一个
            for _ in range(len(self.stack_in)):
```

```
                self.stack_out.append(self.stack_in.pop())
            return self.stack_out.pop()

    def peek(self) -> int:
        """
        获取队列的头部元素（第一个元素），但不移除它。
        Get the front element.
        """
        # 先pop出来最前面一个元素，记下，再加到stack_out回去
        ans = self.pop()
        self.stack_out.append(ans)
        return ans



    def empty(self) -> bool:
        """
        只要in或者out有元素，说明队列不为空
        """
        return not (self.stack_in or self.stack_out)

    # Your MyQueue object will be instantiated and called as such:
    # obj = MyQueue()
    # obj.push(x)
    # param_2 = obj.pop()
    # param_3 = obj.peek()
    # param_4 = obj.empty()
```

225. 用队列实现栈

题目描述

请你仅使用两个队列实现一个**后入先出**（LIFO）的栈，并支持普通栈的全部四种操作：

- `void push(int x)`：将元素 x 压入栈顶。
- `int pop()`：移除并返回栈顶元素。
- `int top()`：返回栈顶元素。
- `boolean empty()`：如果栈是空的，返回 `true`；否则，返回 `false`。

注意

1. 你只能使用队列的标准操作 —— 也就是 `push to back`、`peek/pop from front`、`size` 和 `is empty` 这些操作。
2. 你可以使用 `list`（列表）或者 `deque`（双端队列）来模拟一个队列，只要是标准的队列操作即可。

示例

输入：

```
["MyStack", "push", "push", "top", "pop", "empty"]
[[], [1], [2], [], [], []]
```

输出：

```
[null, null, null, 2, 2, false]
```

解释：

```
MyStack myStack = new MyStack();
myStack.push(1);
myStack.push(2);
myStack.top(); // 返回 2
myStack.pop(); // 返回 2
myStack.empty(); // 返回 False
```

进阶

你能否仅用一个队列来实现栈?

In [ ]:
```
# ChatGPT 用两个list实现，但是意义不大
class MyStack:
    def __init__(self):
        self.queue1 = []  # 主队列，用于存储元素
        self.queue2 = []  # 辅助队列，用于实现 push 的调换逻辑

    def push(self, x: int) -> None:
        """
        将元素压入栈顶
        """
        # 先将新元素放入辅助队列
        self.queue2.append(x)

        # 再将 queue1 的所有元素移到 queue2
        while self.queue1:
```

```
            self.queue2.append(self.queue1.pop(0))

        # 交换 queue1 和 queue2
        self.queue1, self.queue2 = self.queue2, self.queue1

    def pop(self) -> int:
        """
        移除并返回栈顶元素
        """
        if self.empty():
            return None
        return self.queue1.pop(0)

    def top(self) -> int:
        """
        返回栈顶元素
        """
        if self.empty():
            return None
        return self.queue1[0]

    def empty(self) -> bool:
        """
        如果栈为空，返回 True；否则，返回 False
        """
        return len(self.queue1) == 0
```

`deque`（双端队列）是 Python 标准库 `collections` 中的一个类，它提供了高效的双端队列操作，适用于需要快速插入和删除的场景。

特点

- **双端队列**：支持从两端进行高效的插入和删除操作。
- **性能**：
  - 从两端插入和删除操作的时间复杂度为 $O(1)$。
  - 中间位置的访问和操作时间复杂度为 $O(n)$（类似列表）。
- **灵活性**：既可用作栈（后入先出）也可用作队列（先进先出）。

`deque` 的常用方法

1. 初始化

```
from collections import deque

# 创建一个空的 deque
d = deque()

# 创建一个有初始元素的 deque
d = deque([1, 2, 3])
```

---

2. 插入操作

```
# 在右侧添加元素（类似 append）
d.append(4)  # d => deque([1, 2, 3, 4])

# 在左侧添加元素
d.appendleft(0)  # d => deque([0, 1, 2, 3, 4])
```

---

3. 删除操作

```
# 从右侧移除元素（类似 pop）
d.pop()  # 返回 4, d => deque([0, 1, 2, 3])

# 从左侧移除元素
d.popleft()  # 返回 0, d => deque([1, 2, 3])
```

---

4. 访问元素

```
# 获取 deque 的第一个元素（不删除）
first = d[0]  # 返回 1

# 获取 deque 的最后一个元素（不删除）
last = d[-1]  # 返回 3
```

---

5. 其他操作

```
# 获取 deque 的长度
length = len(d)  # 返回 3

# 检查 deque 是否为空
is_empty = len(d) == 0  # 返回 False

# 反转 deque
```

```
d.reverse()  # d -> deque([3, 2, 1])

# 清空 deque
d.clear()  # d -> deque([])
```

用法场景

1. 队列

`deque` 可以用作先进先出的队列：

```
from collections import deque

queue = deque()

# 入队
queue.append(1)
queue.append(2)
queue.append(3)

# 出队
print(queue.popleft())  # 输出 1
print(queue.popleft())  # 输出 2
```

2. 栈

`deque` 可以用作后入先出的栈：

```
from collections import deque

stack = deque()

# 入栈
stack.append(1)
stack.append(2)
stack.append(3)

# 出栈
print(stack.pop())  # 输出 3
print(stack.pop())  # 输出 2
```

3. 滑动窗口

`deque` 可以高效处理固定大小的滑动窗口：

```
from collections import deque

window = deque(maxlen=3)  # 设置最大长度为 3
window.append(1)
window.append(2)
window.append(3)
print(window)  # 输出 deque([1, 2, 3])

window.append(4)
print(window)  # 输出 deque([2, 3, 4]), 自动移除最左端的元素
```

与列表的对比

特性	`deque`	`list`
左端插入和删除	$O(1)$	$O(n)$
右端插入和删除	$O(1)$	$O(1)$
任意位置访问	$O(n)$	$O(1)$（随机访问）
内存效率	更高	较低

总结

- **`deque` 的优势**：
  - 适合需要频繁从两端插入或删除元素的场景。
  - 可以高效实现栈、队列和双端队列的操作。
- **适用场景**：
  - 队列、栈、滑动窗口或需要双端操作的情况。
- **限制**：
  - 不适合需要随机访问的场景，访问任意位置的复杂度为 $O(n)$。

`deque` 在实现栈和队列的操作时，性能优于列表，是标准库中非常实用的数据结构。

In [ ]:
```
# 答案：用两个deque实现

from collections import deque
```

```
class MyStack:

    def __init__(self):
        """
        Python普通的Queue或SimpleQueue没有类似于peek的功能
        也无法用索引访问，在实现top的时候较为困难。

        用list可以，但是在使用pop(0)的时候时间复杂度为O(n)
        因此这里使用双向队列，我们保证只执行popleft()和append()，因为deque可以用索引访问，可以实现和peek相似的功能

        in - 存所有数据
        out - 仅在pop的时候会用到
        """
        self.queue_in = deque()
        self.queue_out = deque()

    def push(self, x:int) -> None:
        # 把新元素放到队列最后
        self.queue_in.append(x)

    def pop(self) -> int:
        """
        输出队列最后一个
        1. 首先确认不空
        2. 因为队列的特殊性，FIFO，所以我们只有在pop()的时候才会使用queue_out
        3. 先把queue_in中的所有元素（除了最后一个），依次出列放进queue_out
        4. 交换in和out，此时out里只有一个元素
        5. 把out中的pop出来，即是原队列的最后一个

        tip：这不能像栈实现队列一样，因为另一个queue也是FIFO，如果执行pop()它不能像
        stack一样从另一个pop()，所以干脆in只用来存数据，pop()的时候两个进行交换
        """
        if self.empty():
            return None

        # 这是一种反转操作?
        for i in range(len(self.queue_in) - 1): # 除了最后一个都移到queue_out，剩下的是栈顶元素
            self.queue_out.append(self.queue_in.popleft()) # 只能用popleft模拟队列 FIFO

        # 然后交换in和out栈，输出栈顶元素
        self.queue_in, self.queue_out = self.queue_out, self.queue_in

        return self.queue_out.popleft()

    def top(self) -> int:
        # 获取栈顶元素， 重新利用pop，最后加进top
        if self.empty():
            return None

        # 利用 pop 获取栈顶元素
        top_element = self.pop()

        # 将栈顶元素重新放回 queue_in
        self.queue_in.append(top_element)

        return top_element

    def empty(self) -> bool:
        return len(self.queue_in) == 0
```

为什么需要交换两个队列?

1. **队列的 FIFO 特性**：
   - 队列的特点是**先进先出**，而栈需要**后入先出**。
   - 因此，每次 `pop` 时需要先将 `queue_in` 的所有元素（除了最后一个）依次移动到 `queue_out`，最后剩下的那个元素就是栈顶元素。
2. **逻辑简化**：
   - 每次 `pop` 或 `top` 后，`queue_in` 和 `queue_out` 的角色发生了转换：
     - `queue_out` 中剩下的元素相当于新栈内容。
     - 为了下一次操作能够继续使用 `queue_in`，需要交换两个队列。
   - 通过交换，不需要频繁拷贝数据或使用额外的标记来区分两个队列的状态。
3. **操作复用**：
   - 通过交换两个队列的角色，避免了将所有元素从 `queue_out` 再移回 `queue_in`。
   - 每次操作后，`queue_in` 始终存储栈的内容，方便后续的 `push` 操作。

只用一个队列实现栈，可以通过每次插入新元素时调整队列中的顺序，使得新插入的元素总是位于队列的最前面，从而模拟栈的“后入先出”（LIFO）特性。

- push(x)：将新元素加入队列末尾，然后将队列中其他所有元素重新移动到队列末尾。
- pop()：直接从队列的前端弹出元素。
- top()：直接获取队列前端的元素。
- empty()：检查队列是否为空。

In [ ]:
```
from collections import deque

class MyStack:
```

```python
    def __init__(self):
        self.queue = deque()  # 单个队列

    def push(self, x: int) -> None:
        """
        将元素压入栈顶
        """
        self.queue.append(x)
        # 这步最重要！！ 队列中除新插入的元素外的所有元素依次移到队列末尾
        for _ in range(len(self.queue) - 1):
            self.queue.append(self.queue.popleft())

    def pop(self) -> int:
        """
        移除并返回栈顶元素
        """
        if self.empty():
            return None
        return self.queue.popleft()

    def top(self) -> int:
        """
        获取栈顶元素
        """
        if self.empty():
            return None
        return self.queue[0]

    def empty(self) -> bool:
        """
        判断栈是否为空
        """
        return len(self.queue) == 0
```

20. 有效的括号

题目描述

给定一个只包括 `()`，`{}`，`[]` 的字符串 s，判断字符串是否有效。

有效字符串需满足：

1. 左括号必须用相同类型的右括号闭合。
2. 左括号必须以正确的顺序闭合。
3. 每个右括号都有一个对应的相同类型的左括号。

示例

示例 1：

```
输入：s = "()"
输出：true
```

示例 2：

```
输入：s = "()[]{}"
输出：true
```

示例 3：

```
输入：s = "(]"
输出：false
```

示例 4：

```
输入：s = "([])"
输出：true
```

提示

- $$1 \leq s.length \leq 10^4$$
- s 仅由括号 `()`、`{}` 和 `[]` 组成。

解题思路

1. 使用**栈**来模拟括号匹配过程：
   - 每遇到一个左括号，压入栈中。
   - 每遇到一个右括号，检查栈顶是否是对应的左括号，若匹配则弹出栈顶，否则字符串无效。
2. 遍历结束后：
   - 栈为空，则字符串有效；

- 栈非空，则字符串无效。

In [ ]:
```python
# 用stack

from collections import deque

class Solution:
    def isValid(self, s: str) -> bool:
        # 用stack来存储
        # [ ], ), ], } ]
        stack = deque()
        for sign in s:
            if sign == '(':
                stack.append(")")
            elif sign == '{':
                stack.append("}")
            elif sign == "[":
                stack.append("]")

            if sign in '])}':
                # stack.pop() # pop不能有parameter、

                # 如果栈为空或栈顶元素不匹配，返回 False
                if not stack or stack.pop() != sign: # 不能pop两次！！在判断中我们已经pop了一次
                    return False

        return len(stack) == 0


# count奇偶不行，因为有括号顺序的限制
```

In [ ]:
```python
# 方法二，使用字典
# 用字典map的方式逻辑一样，但是更简洁清晰!
class Solution:
    def isValid(self, s: str) -> bool:
        stack = []
        mapping = {
            '(': ')',
            '[': ']',
            '{': '}'
        }
        for item in s:
            if item in mapping.keys():
                stack.append(mapping[item])
            elif not stack or stack[-1] != item:
                return False
            else:
                stack.pop()
        return True if not stack else False
```

1047. 删除字符串中的所有相邻重复项

题目描述

给出由小写字母组成的字符串 s，重复项删除操作会选择两个相邻且相同的字母，并删除它们。

在 s 上反复执行重复项删除操作，直到无法继续删除。

在完成所有重复项删除操作后返回最终的字符串。答案保证唯一。

示例

输入：

```
s = "abbaca"
```
输出：

```
"ca"
```
解释：

- 在字符串 "abbaca" 中，可以删除 "bb"，由于两字母相邻且相同，这是此时唯一可以执行删除操作的重复项。
- 之后得到字符串 "aaca"，其中又只有 "aa" 可以执行重复项删除操作，所以最后的字符串为 "ca"。

In [ ]:
```python
# 用栈非常巧妙，因为他模拟了最终结果

from collections import deque

class Solution:
    def removeDuplicates(self, s: str) -> str:
        stack = deque()
        for l in s:
            if len(stack) > 0:
                if stack[-1] == l:
                    stack.pop()
                else:
```

```
                    stack.append(l)
            else:
                stack.append(l)

        string = "".join(stack)
        return string
```

知识点：

- stack.length没有
- len(stack)可以
- stack[-1]可以，但在用之前要确保stack不为空
- string = "".join(stack) 也可以用