2-10 The mutual realization of stack and queue

杜星亮*(stardustdl@163.com)*

2018/8/1

本文介绍使用栈实现队列和使用队列实现栈的方法。为使问题更加有意义，我们将尽量选择一个较优的实现方法。[1]

[1] 如发现错误，欢迎指正，网页版链接

第*0*节：问题引入

- 栈：先进后出的线性结构，仅允许对栈顶进行添加（push），删除（pop），访问（peek）操作，空间复杂度线性，单次操作时间复杂度为常数
- 队列：先进先出线性结构，仅允许对队尾进行添加（enqueue）操作，以及对队首进行删除（dequeue），访问（peek）操作，空间复杂度线性，单次操作时间复杂度为常数
- 需要解决的问题，在使用常数个额外空间的条件下：
  - 使用两个栈尽可能高效地实现一个队列
  - 使用两个队列尽可能高效地实现一个栈

第*1*节：使用栈实现队列

思路

考虑到栈先进后出与队列先进先出的特点，使用一个栈A 作为队列尾，数据从这里流入；使用另一个栈B 作为队列头，数据从这里流出。我们要求先流入的数据先流出，通过将栈A 中的元素不断弹出，并压入栈B，利用先进后出特性，B 的出栈顺序即A 出栈顺序的逆序，而A 出栈顺序为其入栈顺序的逆序，故B 的出栈顺序为A 的入栈顺序，即达到先进先出的效果。综上，给出实现的伪代码。

栈能将输入逆序这一点很重要，之后我们还将用到

复杂度分析

- 空间复杂度：栈的空间复杂度是线性的，而且这里队列中的每一个数据仅会在两个栈中的某一个中存在，故此实现的空间复杂度为线性
- 时间复杂度：注意到每个数据从入队到出队只会经历：进入A，离开A，进入B，离开B 四次移动，且每次移动复杂度为常数，故均摊复杂度为常数。

Algorithm 1 队列三个操作的实现

```
    ▷ Move elements in A to B
1:  procedure SWAP(A, B)
2:      while A ≠ ∅ do
3:          value ← POP(A)
4:          PUSH(B, value)

5:  procedure ENQUEUE(A, B, value)
6:      PUSH(A, value)

7:  procedure DEQUEUE(A, B)
8:      if B = ∅ then
9:          SWAP(A, B)
        return POP(B)

10: procedure PEEK(A, B)
11:     if B = ∅ then
12:         SWAP(A, B)
        return PEEK(B)
```

第2节：使用队列实现栈

思路

这里遇到了个难题：利用栈可以很容易地支持翻转操作，但队列无法直接对输入序列进行顺序的改变。

队列中一个重要特点是，我们可以通过不断删除队列头，并将其放入队列尾，实现在不影响顺序的前提下对队列中每个元素的访问，访问一遍后，我们仍可以很容易恢复到最初的队列状态。但这一操作的弊端是，我们访问某个元素，必须将其前面的所有元素出队，这一操作的时间复杂度是最坏情况下是线性的。 为下文叙述方便，将上述操作定义为过程“循环出入队”，即将队首出队后入队，实现队列滚动。时间复杂度由以上分析，为$O(|Q|)$。我们使用一个类似缓冲池的技巧：设两个队列Q_s, Q_a，Q_a 用于存储靠近栈顶的一部分元素，Q_s 用于存储其余的元素。其中Q_a 有可变的容量上限$cap(Q_a)$，Q_s 容量无限制。 接下来，我们依次实现栈的三个操作：

既然我们能访问所有元素了，那只要访问最后一个就是先入后出了，很简单嘛，可是队列“滚”的次数太多了（为线性）...

你可以把Q_s 看成内存（主存），把Q_a 看成CPU 中的高速缓存

- 入栈操作
  - 若Q_a 不满，直接入队到Q_a 。时间复杂度：$O(1)$
  - 若Q_a 满，将Q_a 出队，并将队列头入队到Q_s。然后将待入栈元素入队到Q_a。时间复杂度：$O(1)$
- 出栈操作
  - 若Q_a 非空，对Q_a 循环出入队，使得原队尾在队头，返回队尾，并出队。时间复杂度：$O(|Q_a|)$
  - 若Q_a 空，对Q_s 循环出入队，使得原队尾部的$|Q_a|+1$ 个元素出队，返回队尾，其余元素进入Q_a，顺序不变。时间复杂

度：$O(|Q_s|)$

- 访问栈顶操作
  - 若Q_a 非空，对Q_a 循环出入队，使得原队尾在队头，返回队尾，并恢复最初顺序，时间复杂度：$O(|Q_a|)$
  - 若Q_a 空，对Q_s 循环出入队，使得原队尾部的$|Q_a|$ 个元素出队，返回队尾，所有元素进入Q_a，顺序不变。时间复杂度：$O(|Q_s|)$

似乎看不出有什么改进？是的，因为我们还没指定$cap(Q_a)$

为让这个实现变得有效，我们先来分析其时间复杂度来自哪里：**每个元素**从入栈到出栈经历了：

1. 压入：$O(1)$
2. （可选）从Q_a 到Q_s：$O(1)$
3. （可选）从Q_s 到Q_a：$O(|Q_s|/|Q_a|)$（最坏情况）
4. 弹出以下两种二选一：
   - 从Q_s 出队：$O(1)$
   - 从Q_a 出队：$O(|Q_a|)$

接下来考虑最坏情况，那么每个元素经历入队出队共计$O(|Q_s|/|Q_a| + |Q_a|)$.

- 由$|Q_s|/|Q_a| + |Q_a| \geq 2\sqrt{|Q_s|}$ 当且仅当$|Q_s|/|Q_a| = |Q_a|$ 即$|Q_a| = \sqrt{|Q_s|}$.
- 故最低为$O(\sqrt{|Q_s|})$

故当我们将Q_a 的容量限制在$\sqrt{|Q_s|}$ 时，对于n 个元素的栈，有$|Q_s| \leq n$ 总复杂度估计为$O(n\sqrt{n})$. 平均每次操作复杂度$O(n\sqrt{n}/(2n)) = O(\sqrt{n})$，由此可见，性能的确有较大提高。

综上，给出实现的伪代码。

如果你像上面那样类比了计算机中的组件，那么我们的算法做的，就可以看成从内存中读取一段，放入高速缓存中，利用高命中率来减少我们对相对低速的内存的访问

复杂度分析

- 空间复杂度：队列的空间复杂度是线性的，而且这里栈中的每一个数据仅会在两个队列中的某一个中存在，故此实现的空间复杂度为线性
- 时间复杂度：通过思路中的分析，单次操作均摊复杂度为$O(\sqrt{n})$。

更多讨论

- 时间复杂度下限的讨论
- 队列实现栈的CSharp 代码实现

Algorithm 2 栈三个操作的实现

```
1: procedure PUSH(Q_a, Q_s, value)
2:     if |Q_a| ≥ √|Q_s| then
3:         ENQUEUE(Q_s,DEQUEUE(Q_a))
4:     ENQUEUE(Q_a, value)
5: procedure POP(Q_a, Q_s)
6:     if Q_a = ∅ then
7:         size ← √|Q_s|, i ← 0
8:         Roll Q_s until the last size + 1 elements are at the front
9:         while i < size do
10:            ENQUEUE(Q_a,DEQUEUE(Q_s))
11:            i ← i + 1
           return DEQUEUE(Q_s)
12:    else
13:        Roll Q_a until the last element is at the front return
   DEQUEUE(Q_a)
14: procedure PEEK(A, B)
15:    if Q_a = ∅ then
16:        size ← √|Q_s|, i ← 0
17:        Roll Q_s until the last size elements are at the front
18:        while i < size do
19:            ENQUEUE(Q_a,DEQUEUE(Q_s))
20:            i ← i + 1
21:        value ← DEQUEUE(Q_s)
22:        ENQUEUE(Q_a, value) return value
23:    else
24:        Roll Q_a until the last element is at the front
25:        value ← DEQUEUE(Q_a)
26:        ENQUEUE(Q_a, value) return value
```