Monotonic Queue

Stephen Huan
Edited by: Udbhav Muthakana

January 8, 2020

1 Definition

A ***monotonic queue*** is a queue with $O(1)$ inserts and pops on both sides, but with the additional property of computing the minimum element in $O(1)$.

2 Implementation

2.1 Min Stack

Consider a min stack. Since a stack is first in last out, we only need to propagate a minimum element "upward" in the stack - to elements added after it. Let each number in the stack be represented by a tuple of (value, minvalue). Maintain the min value as follows: For an empty stack, a new value will clearly be the minimum, so add the tuple (value, value). If the stack is not empty, compare the new value with the current minimum (accessing the 2nd number in the tuple at the top of the stack), and make the smaller of the two the minimum value. When removing elements, just pop as normal.

Thus, at any given moment the stack can return its minimum element by accessing the top of the stack in constant time, while still maintaining $O(1)$ addition and removal.

In Python:

```
class MinStack:

    def __init__(self, f=min): self.stk, self.f = [], f

    def empty(self): return len(self.stk) == 0

    def append(self, val):
      self.stk.append((val, \
      self.f(val, self.stk[-1][1]) if not self.empty() else val))

    def pop(self): return self.stk.pop()[0]

    def min(self): return self.stk[-1][1]
```

2.2 Min Queue

Now, consider simulating a queue using two stacks. Intuitively, a stack will "reverse" elements while a queue will keep them in the same order. For example, adding 1, 2, 3 to a stack will yield 3, 2, 1 when popped, while a queue will yield 1, 2, 3.

Using two stacks reverses the reverse, preserving the original order. Designate one stack as the "in-stack" and the other as the "out-stack". When adding elements, add them to the in-stack. When removing elements, check whether the out-stack is empty or not. If the out-stack is empty, pop off everything on the in-stack add them to the out-stack. Otherwise, pop off an element from the out-stack. By only popping off elements from the in-stack to the out-stack if the out-stack is empty, we can guarantee continuity in the reversal. For example, consider the following sequence of pushes and pops:

add 1, add 2, pop, add 3, pop, pop

After the first two additions, the in-stack contains [1, 2] and the out-stack is empty. Then, on the pop, the in-stack empties into the out-stack, which is now [2, 1]. We then pop the out-stack to return 1 and add 3 to the in-stack. At this point, our in-stack is [3], and our out-stack is [2]. On the next pop, the out-stack is emptied. Finally, on the last pop, the in-stack again empties into the out-stack, which then pops its only element.

If we use two min stacks to implement a queue, the minimum element in the queue is just the minimum element between the stacks. Since we can access the minimum element of a min stack in $O(1)$, we can find the minimum in our min queue in $O(1)$. We still have constant time addition to the queue, since we just add it to the in-stack. When popping an element off, it is processed at most two times (entering the in-stack and being transferred to the out-stack). Thus, the amortized work in popping is $O(1)$.

```
class MinQueue:

    def __init__(self, f=min):
      self.instk, self.outstk, self.f = MinStack(f), MinStack(f), f

    def enque(self, val): self.instk.append(val)

    def deque(self):
        if self.outstk.empty():
            while not self.instk.empty(): self.outstk.append(self.instk.pop())
        return self.outstk.pop()

    def min(self):
        if not self.instk.empty() and not self.outstk.empty():
            return self.f(self.instk.min(), self.outstk.min())
        elif self.instk.empty(): return self.outstk.min()
        elif self.outstk.empty(): return self.instk.min()
```

3 Past Lectures

1. "Monotonic Queues and USACO Review" (Daniel Wisdom, 2019)

4 Works Cited

1. Richard Zhan