=============================================================================

CSC 263 Tutorial 6 Winter 2019

=============================================================================

In this tutorial, we explore a well-known trick that uses two Stacks S1 and S2 to simulate the operations of a Queue Q. Consider the following implementations of Enqueue and Dequeue.

def Enqueue(Q, x):

if Size(S1) >= 12 and IsEmpty(S2):

while not IsEmpty(S1):

Push(S2, Pop(S1))

Push(S1, x)

return

def Dequeue(Q):

if IsEmpty(S2):

if IsEmpty(S1):

error "Dequeuing from an empty queue!"

else:

while not IsEmpty(S1):

Push(S2, Pop(S1))

return Pop(S2)

For your analysis, assume that each Push operation takes 2 units of time, and

each Pop operation takes 3 units of time. Each IsEmpty or Size operation takes 0 units of time. The time taken by any other pseudo-code operation is ignored.

(a) First, convince yourself and your partners that this is a correct

implementation of the Queue ADT. Trace some examples.

(b) Consider the sequence of operations that consists of 50 Enqueue operations followed by 50 Dequeue operations. Use the aggregate method to compute the amortized cost per operation for this sequence.

The operations can be categorized into the following types:

* “Short” Enqueue: A simple Push which costs $2.
* “Long” Enqueue: A Push following the migration of 12 elements from S1 to S2, whose cost is 5×12+2 =  $62.
* “Short” Dequeue: A simple Pop which costs $3.
* “Long” Dequeue: A Pop following the migration of all elements from S1 to S2, whose cost is 5k + 3  (with k being the number of elements to migrate).
* The sequence of 50 Enqueue and 50 Dequeue operations then looks as follows:
* 12 short Enqueues
* 1 long Enqueue
* 37 short Enqueues (因为要再long enqueue的情况已经不存在了，因为他还有一个条件是IsEmpty(S2)，第一次long enqueue以后S2已经不是empty了，所以之后的都是short enqueue了)
* 12 short Dequeues（之前12个short enqueue放进去的12个element）
* 1 long Dequeue with the migration of 38 elements（是之前的一个long enqueue其中包括一次short enqueuer，和37个short enqueuer，共38个element）
* 37 short Dequeues（剩下的所有element）

The total cost is: Therefore the amortized cost per operation is:

12·$2+1·$62+37·$2+12·$3+1·($5·38+3)+37·$3=$500.

$500/100 = $5.

(c) Now consider any sequence of m Enqueue and Dequeue operations. Use the

accounting method to derive an upper-bound on the amortized cost per

operation.

这里要注意的是我们每个operation的cost已经不是$1了，要根据这个情况来考虑。

Using the accounting method, we charge $10 for each Enqueue operation and $0 for each Dequeue operation. Imagine that this $10 is “attached” to the element that was Enqueued: $2 pays for pushing the element into S1, $5 for migrating the element from S1 to S2 (1 Push + 1 Pop), and $3 for popping the element out of S2. This $10 per element is enough to pay for all operations since each element is pushed, migrated and popped at most once during any sequence of operations. Hence, the amortized cost per operation for any sequence is *at most* 10 units of time (maybe less depending on the sequence).

another two amortized questions

1. Recall the lecture example of expanding dynamic arrays during a sequence of APPEND operations. Our analysis showed that, by doubling the size of the array when it is full (i.e., expansion factor = 2), the amortized cost per operation is tightly upper-bounded by 3. Now we want to change the expansion factor so that the amortized cost per operation becomes tightly upper-bounded by 7. What should the new expansion factor be?  
  
Dan says: so each element gets 7 dollars here. 1 dollar is used up right away when the element gets appended. Another 1 dollar is used to copy the element itself when the array expands. That leaves 5 dollars for copying other elements over. This means that each new element can copy over 5 old elements. Say that the array just expanded, and we have c old elements. We need c/5 = 0.2c new elements before we have enough money to expand again. At that time, the array will have c+0.2c = 1.2c elements. The expansion factor is therefore 1.2.

这里可以用我们总结出来的公式：expansion factor = 1/(charge -2) + 1,把charge带入7就好。  
  
  
2. In the lecture we learned that, if we expand the dynamic array by doubling the size when it is full, the amortized cost per operation of a sequence of APPEND operations is 3. If we change the expansion rule to multiply the size by 1.1 when it is full, i.e., size = size \* 1.1, what is the amortized cost per operation?  
  
Dan says: above, we were given the desired amortized cost and asked to find an expansion factor. Here, we're given an expansion factor and asked to find the amortized cost.  
  
Multiplying the size by 1.1 means that each new element will have to copy over 10 old（？？？？？） elements. e.g. if the array just expanded and we had 20 elements, we fill up the array again when the array has 22 elements (20\*1.1). Each of the 2 new elements has to copy over 10 of the old elements, plus itself, for a total of 11. But remember that the initial Append of the element costs 1 dollar too, so we need 12 dollars per element.

这里可以用我们总结出来的公式：expansion factor = 1/(charge -2) + 1,把expansion factor带入12就好。

note: the dollar analogy is actually referring to the runtime. ($3 means 3 steps)

note: in this basic case, **$4 still works, but it’s not the tightest bound.**

more practice:

1. find expansion factor such that $7 is the minimum cost per element
2. find amortized cost per operation (cost $), such that expansion factor is 1.1.
3. find amortized cost per operation (cost $), such that expansion factor is 1.5.

这几题都是用我们的公式就好