CS545 Spring, 2015

Homework Assignment #4
Due: April 16
Student: Shuo Yang

---

1. Let two stacks be S_{rear} and S_{front} where S_{rear} is used for putting elements to the rear of queue Q and S_{front} is used for removing elements on the front of queue Q. To implement $Put(x, Q)$, we just push x to S_{rear} such that the tail of queue would be on top of S_{rear}. To implement $Get(Q)$, we pop element from S_{front}, if S_{front} is empty but S_{rear} is not empty, we first pop every element off S_{rear} and push them to S_{front} such that the front of queue would be on top of the S_{front}, then do the pop.

   Pseudo code

   Function $Put(x, Q)$
       $push(x, S_{rear})$

   Function $Get(Q)$
       if S_{front} is not empty
           return $pop(S_{front})$
       else if S_{rear} is not empty
           while S_{rear} is not empty
               $x = pop(S_{rear})$
               $push(x, S_{front})$
           return $pop(S_{front})$
       else
           print "Empty Queue"

   Amortized Analysis

   We will use the number of basic push and pop to measure cost.

   For each $i = 1, 2, \cdots, n$, let a_i be the amortized cost of the ith operation, t_i be the actual cost for ith operation, and D_i be the data structure that results after applying the ith operation to data structure D_{i-1}. We start with D_0.

   Let the number of elements in the stack S_{rear} be s. We define the potential function Φ be $2s$. For the empty queue D_0 with which we start, we have $\Phi(D_0) = 0$. Since the number of elements in the stack is never negative, the queue D_i that results after the i_{th} operation has non-negative potential, thus,

$$\begin{aligned} \Phi(D_i) &\geq 0 && (1) \\ &= \Phi(D_0) && (2) \end{aligned}$$

   The total amortized cost of n operations with respect to Φ therefore represents an upper bound on the actual cost.

   Suppose the ith operation on a queue with s elements in the stack S_{rear} is Put, then the amortized cost is:

$$\begin{aligned} a_i &= t_i + \Phi(D_i) - \Phi(D_{i-1}) && (3) \\ &= 1 + 2(s+1) - 2s && (4) \\ &= 1 + 2 && (5) \\ &= 3 && (6) \end{aligned}$$

t_i is 1 because *Put* only took 1 basic push. Potential before the operation is $2s$ and potential after the operation is $2(s+1)$ since the size of the stack S_{rear} grows by 1. Thus the change of potential is 2.

If it is a *Get* operation, there are two cases to consider:

(a) stack S_{front} is not empty, then the amortized cost is:

$$\begin{aligned} a_i &= t_i + \Phi(D_i) - \Phi(D_{i-1}) && (7)\\ &= 1 + 2s - 2s && (8)\\ &= 1 + 0 && (9)\\ &= 1 && (10) \end{aligned}$$

Again, *Get* in this case only took 1 basic pop, so t_i is 1. The potential didn't change since the size of stack S_{rear} didn't change.

(b) stack S_{front} is empty, then the amortized cost is:

$$\begin{aligned} a_i &= t_i + \Phi(D_i) - \Phi(D_{i-1}) && (11)\\ &= (s + s + 1) + 0 - 2s && (12)\\ &= 2s + 1 - 2s && (13)\\ &= 1 && (14) \end{aligned}$$

In this case, *Get* operation took s basic pop and s basic push to remove all elements in S_{rear} into S_{front}, and 1 basic pop to get the element on the front of queue. Since after the operation, S_{rear} would be empty, thus the change of potential is $-2s$.

The amortized cost for each of the two operations is $O(1)$, and thus of total cost of a sequence of n operations is $O(n)$. Since we've already shown that the total amortized cost of n operations is an upper bound on the total actual cost. The worst-case cost of n operations is therefore $O(n)$.

2. We will use an unsorted array A to implement these two operations. Let n be the size of A. Initially, $n = 0$.

Pseudo code

Function $Insert(x, S)$
  $n := n + 1$
  $A[n] := x$

Function $DeleteLargerHalf(S)$
  use the worst-case linear time selection algorithm to find the median of A.
  partition the array A around the median.
  remove the elements from the larger half of the partitioned array A.
  $n := n - \lceil n/2 \rceil$ // reset the size of A

$Insert$ takes constant time while $DeleteLargerHalf$ takes $O(n)$ time since finding the median takes linear time, and so do partitioning the array and removing elements from the larger half.

Amortized Analysis

We will use the number of basic operations that take constant amount of time to measure the cost such that the run time would be Θ of the number of basic operations. Since $Insert$ takes two basic operations (incrementing n and assigning x to $A[n]$), its actual cost would be 2. Since $DeleteLargerHalf$ takes $O(n)$ time, let its actual cost be cn where c is some positive constant.

The following table shows the real time and amortized time for each operation.

operation	actual time t_i	amortized cost a_i
$Insert$	2	2+2c
$DeleteLargerHalf$	cn	0

For $Insert$, we use 2 unit out of 2+2c units to pay the actual cost and store the remaining $2c$ units as credit for each inserted element. For $DeleteLargerHalf$, we use c unit of credit stored on each element to pay for the actual cost. This leaves c unit of credit on each element after finding the median and partitioning the array. When deleting the larger half, we redistribute the c unit of credit stored on each deleted element to the remaining elements. Thus, there are always $2c$ unit of credit stored on each element so we can pay for future $DeleteLargerHalf$ operations.

Since each element in the array has $2c$ unit of credit on it, and the size of array is always non-negative, we have ensured that the amount of credit is always non-negative. Thus, for any sequence of n $Insert$ and $DeleteLargerHalf$ operations, the total amortized cost is an upper bound on the total actual cost.