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 which is an empty queue.

   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 (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 at most cn where c is some positive constant.

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

operation	actual cost 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.

3. Let D_i be the heap with n_i elements after the ith operation. Since both *Extract* and *Insert* take $O(\lg n)$ time, let k be some positive constant such that both operations take at most $k \lg n$, where $n = max(n_i, n_{i-1}$. So now we can use $k \lg n$ to measure the real time of each operation.

   We define the potential function be:

$$\Phi(D_i) = \sum_{x \in D_i} (1 + k \cdot depth_i(x)) \tag{15}$$

$$= n_i + k \sum_{x \in D_i} depth_i(x) \tag{16}$$

   where $depth_i(x)$ is the depth (the number of edges from the root to the node) of element x in D_i.

   Initially, the heap is empty, so $\Phi(D_0) = 0$. And since k is postive and $depth_i(x)$ is non-negative, we always have $\Phi(D_i) \geq 0$.

   Suppose the ith operation is *Extract*, the change of potential is:

$$\Delta\Phi = \Phi(D_i) - \Phi(D_{i-1}) \tag{17}$$

$$= (n_i + k \sum_{x \in D_i} depth_i(x)) - (n_{i-1} + k \sum_{x \in D_{i-1}} depth_{i-1}(x)) \tag{18}$$

$$= (n_i - n_{i-1}) + k(\sum_{x \in D_i} depth_i(x) - \sum_{x \in D_{i-1}} depth_{i-1}(x)) \tag{19}$$

$$= -1 + k(-\lg n_{i-1}) \tag{20}$$

$$= -1 - k \lg n_{i-1} \tag{21}$$

   since the size of heap after *Extract* drops by 1 and the sum of depth of all elements in D_i drops by the depth of the last element in D_{i-1}, which is $\lg n_{i-1}$, so the potential decreases by $(1 + k \lg n_{i-1})$.

So, the amortized time for *Extract* is:

$$\begin{aligned} a_i &= t_i + \Delta\Phi && (22)\\ &\le k \lg n_{i-1} + (-1 - k \lg n_{i-1}) && (23)\\ &\le k \lg n_{i-1} - 1 - k \lg n_{i-1} && (24)\\ &\le -1 && (25)\\ &= O(1) && (26) \end{aligned}$$

since the actual time takes for *Extract* is at most $k \lg n_{i-1}$ because $n_{i-1} > n_i$, thus the amortized time is $O(1)$.

Suppose the ith operation is *Insert*, the change of potential is:

$$\begin{aligned} \Delta\Phi &= \Phi(D_i) - \Phi(D_{i-1}) && (27)\\ &= (n_i + k \sum_{x \in D_i} depth_i(x)) - (n_{i-1} + k \sum_{x \in D_{i-1}} depth_{i-1}(x)) && (28)\\ &= (n_i - n_{i-1}) + k(\sum_{x \in D_i} depth_i(x) - \sum_{x \in D_{i-1}} depth_{i-1}(x)) && (29)\\ &= 1 + k \lg n_i && (30) \end{aligned}$$

since the size of heap after *Insert* increases by 1 and the sum of depth of all elements in D_i increases by the depth of the last element in D_i, which is $\lg n_i$, so the potential increases by $(1 + k \lg n_i)$.

So, the amortized time for *Insert* is:

$$\begin{aligned} a_i &= t_i + \Delta\Phi && (31)\\ &\le k \lg n_i + (1 + k \lg n_i) && (32)\\ &\le 1 + 2k \lg n_i && (33)\\ &= O(\lg n) && (34) \end{aligned}$$

since the actual time takes for *Insert* is at most $k \lg n_i$ because $n_i > n_{i-1}$, thus the amortized time is $O(\lg n)$.

4. (a) We can implement *Find* by doing binary search on each array in D that is *full* since individual arrays are sorted.

Pseudo code

```
Function $Find(k, D)$
    for $i := 1$ to $\lceil \lg(n+1) \rceil$
        if bit $i$ is 1 in binary representation of $n$: // $A_i$ is full
            search key $k$ in $A_i$ using binary search.
            if $k$ is found, return the item $x$ associated with it;
            otherwise continue to the next array.
```

The worst case for *Find* is when all arrays $A_1, \cdots, A_l$ are full, where $l = \lceil \lg(n+1) \rceil$, then the total time taken is:

$$T(n) = \Theta(\sum_{i=1}^{l} \log(2^{i-1})) \tag{35}$$
$$= \Theta(\sum_{i=1}^{l}(i-1)) \tag{36}$$
$$= \Theta(\frac{l(l-1)}{2}) \tag{37}$$
$$= \Theta(l^2) \tag{38}$$
$$= \Theta(\lceil \lg(n+1) \rceil^2) \tag{39}$$
$$= \Theta((\log(n))^2) \tag{40}$$

Thus worst-case time for $Find$ is also $O((\log(n))^2)$.

(b) To implement $Insert$, we first create a new sorted array B_1 with size of 1 that contains the element (k, x) to be inserted. Then we check to see if A_1 is empty, if it is, just replace A_1 with B_1, otherwise we merge sort A_1 and B_1 into a new sorted array B_2. Then check if A_2 is empty, if it is, just replace A_2 with B_2, otherwise we merge sort A_2 and B_2 and continue to check the emptiness of A_3 until we find a empty array to replaced. Notice that A_i is of size 2^{i-1} and merging A_i with another array of the same size results in a new array of size 2^i which is A_{i+1}. Assume that merge 2 sorted arrays of the same size m into one sorted array takes $2m$ time. The worst-case time for $Insert$ is when array $A_1, \cdots, A_{l-1}$ are all full so we have to merge sort all of them with B into A_l. This takes:

$$T(n) = 2(\sum_{i=1}^{l-1} 2^{i-1}) \tag{41}$$
$$= 2(2^{l-1} - 1) \tag{42}$$
$$= 2^l - 2 \tag{43}$$
$$= 2^{\lceil \lg(n+1) \rceil} - 2 \tag{44}$$
$$= \Theta(n) \tag{45}$$

Thus the worst-case time for $Insert$ is $\Theta(n)$.

Amortized Analysis

We will the aggregation method to compute the total cost for a sequence of n $Insert$ operations, starting with the empty data structure D_0. For D_i with n elements, let the binary representation of n be $n_k, n_{k-1}, \cdots, n_1$ and let j be the position of the right most 0 bit in $n_k, n_{k-1}, \cdots, n_1$ such that $n_{j-1}, \cdots, n_1$ are all 1s. Therefore, the cost of inserting the $(n+1)$th element would be:

$$T(n) = 2(\sum_{i=1}^{j-1} 2^{i-1}) \tag{46}$$
$$= 2(2^{j-1} - 1) \tag{47}$$
$$= 2^j - 2 \tag{48}$$
$$= O(2^j) \tag{49}$$

And we have $j = 1$ for half time since each time n increases, its the 1st bit will flip, and half of them is flip from 0 to 1, and $j = 2$ for a quarter of time, and so on. So there are at most

$\lceil n/2^{j-1} \rceil$ insertions for each value of j. Thus the total cost of n operations is:

$$O(\sum_{j=1}^{\lceil \lg(n+1) \rceil} \lceil n/2^{j-1} \rceil 2^j) = O(n \log n) \tag{50}$$

Therefore the amortized cost for each of *Insert* operation $O(\log n)$.

5. (a) Suppose T is a n-node binary search tree rooted at node r. To make it 1/2 balanced, we first do an in-order traverse starting at the root r and output all n elements in an array A. Since T is a binary search tree, elements in A will be sorted. This requires $\Theta(n)$ space and time. Then we pick the median of array A and use it as the root to rebuild the tree. This guarantees that the new tree is 1/2 balanced. Rebuilding a tree is a recursive process and the recurrence equation for the run time is $T(n) = 2T(n/2) + 1 = \Theta(n)$. And the extra space needed for the new tree is also $\Theta(n)$. Thus T can be made 1/2 balanced in $\Theta(n)$ time and $\Theta(n)$ space.

   (b) We perform a *Find* operation starting at the root recursively to subtrees, the worst case is when we split in a subtree with m nodes and recurse on its subtree with αm nodes since the number of nodes in a subtree is bounded by α. So the recurrence equation is:

$$\begin{aligned} T(n) &\le T(\alpha n) + 1 \quad (51) \\ &= O(\log_{1/\alpha} n) \quad (52) \\ &= O(\log n) \quad (53) \end{aligned}$$

   (c) First, we prove that for a 1/2-balanced tree, $d(x) < 2$ for all node x in the tree.

Proof. Prove by contradiction. Assume that for some node x, $d(x) \ge 2$, then without loss of generality, assume that the left subtree is bigger, then we have $s(l(x)) - s(r(x)) \ge 2$. Further, we know that the sum of size of left subtree and size of right subtree must be $s(x) - 1$:

$$s(l(x)) + s(r(x)) + 1 = s(x) \tag{54}$$

Replacing $s(r(x))$ with $s(x) - s(l(x)) - 1$, we have:

$$\begin{aligned} s(l(x)) - (s(x) - s(l(x)) - 1) &\ge 2 \quad (55) \\ s(l(x)) - s(x) + s(l(x)) + 1 &\ge 2 \quad (56) \\ 2s(l(x)) &\ge s(x) + 1 \quad (57) \\ s(l(x)) &\ge \frac{1}{2}s(x) + 1/2 \quad (58) \end{aligned}$$

But this contradicts to the fact that the tree is 1/2-balanced. □

Therefore, after rebalancing, the potential $\Phi(T) = 0$ since no node x exists such that $d(x) \ge 2$. Suppose after *Insert* or *Delete*, node x is the highest node that is not α-balanced and the subtree rooted at x is of size m. Without loss of generality, assume that the left subtree is bigger, then we must have $s(l(x)) > \alpha m$. Since $s(l(x)) + s(r(x)) + 1 = s(x) = m$, we have $s(r(x)) = m - 1 - s(l(x)) < m - 1 - \alpha m = (1 - \alpha)m - 1$. Therefore,

$$\begin{aligned} d(x) &= s(l(x)) - s(r(x)) \quad (59) \\ &> \alpha m - ((1 - \alpha)m - 1) \quad (60) \\ &= (2\alpha - 1)m + 1 \quad (61) \end{aligned}$$

Thus we can bound the total potential before rebalancing the tree as:

$$\Phi(T) > \frac{1}{2\alpha - 1}((2\alpha - 1)m + 1) \tag{62}$$

$$= m + \frac{1}{2\alpha - 1} \tag{63}$$

Since the potential after rebalancing the tree is 0, thus the change of potential is $-m - \frac{1}{2\alpha-1}$. Since the actual cost for rebalancing is m, therefore the amortized cost is $O(1)$.

(d) Since *Find* takes $O(\log n)$ worst-case time for a α-balanced tree, *Insert* or *Delete* also takes $O(\log n)$ worst-case time. Further, when we insert or delete a node x, we can only change the nodes that are on the path from the node x to the root. For *Insert*, the worst case would be we increase $d(i)$ for each node i in the path by 1, and there are $O(\log n)$ such nodes. Therefore the potential could increase by $\frac{1}{2\alpha-1}O(\log n) = O(\log n)$. Thus, the amortized cost would be the actual cost plus the change of potential: $O(\log n) + O(\log n) = O(\log n)$.