336-6

avivaid2018@u.northwestern.edu

May 2016

1 Problem 1

Consider an ordinary binary min-heap data structure with n elements supporting the instructions INSERT and EXTRACT-MIN in $O(lg(n))$ worst-case time. Give a potential function ϕ such that the amortized cost of INSERT is $O(lg(n))$ and the amortized cost of EXTRACT-MIN is $O(1)$ and show that it works.

Consider the potential function that maps each data structure D_i with a real number $\phi(D_i)$ such that $\phi(D_i) = O(nlg(n)) = knlg(n)$ where k is a positive constant and n is the number ofitems in the data structure.
The amortized cost $\hat{c_i}$ of the i^{th} operation is:
$\hat{c_i} = c_i + \phi(D_i) - \phi(D_{i-1})$
The actual cost c_i of an operation i is $O(lg(n)) = klg(n)$ for INSERT and EXTRACT-MIN, our two operations.
Consider the initially empty heap D_0 for which the potential $\phi(D_0) = 0$ by our definitions.

Consider the i^{th} operation EXTRACT-MIN. By our above definitions we have:
$\hat{c_i} = klog(n) + \phi(D_i) - \phi(D_{i-1})$
where D_i and D_{i-1} are binary heaps containing n and $n+1$ elements respectively. Therefore,
$\hat{c_i} = klog(n) + knlg(n) - k(n+1)lg(n+1) = klog(n) + knlg(n) - k(n+1)lg(n+1)$ $= klg(n) - klg(n+1) + knlg(n) - knlg(n+1) = (k - kn)(lg(\frac{n}{n+1}))$ which converges to a constant as n tends to ∞ Therefore,
$\hat{c_i} = O(1)$

Consider the i^{th} operation INSERT. By our above definitions we have:
$\hat{c_i} = klog(n) + \phi(D_i) - \phi(D_{i-1})$
where D_i and D_{i-1} are binary heaps containing n and $n-1$ elements respectively. Therefore,
$\hat{c_i} = klog(n) + knlg(n) - k(n-1)lg(n-1) = klog(n) + knlg(n) - k(n-1)lg(n-1)$ $= klg(n) + klg(n-1) + knlg(n) - knlg(n-1) = (k + kn)(lg(\frac{n}{n-1}))$ Therefore,
$\hat{c_i} = klog(n_i) + klog(n_i + 1) = O(log(n))$

2 Problem 1b

No, there is no possible such potential function as since by definition for a particular element to be extracted from a heap- it needs to be inserted first. Since, in analogy to the accounting method (we need to build up credit), we need $\phi(D_i) > \phi(D_0)$ for all i. The only way we can guarantee this is if insert is more expensive than extract min in amortized analysis. Consider, an empty heap- we can't even build up potential if we call extract min before insert! We have the potential equation $\hat{c_i} = klog(n) + \phi(D_i) - \phi(D_{i-1})$ which has a positive actual cost of insertion. So, to get a better amortized cost, the difference in potentials will need to negative. But we cannot ensure this. Consider intially empty stack with initial potential 0, when we insert the first element- the difference in potential is greater than 0 already!
Thusly, there cannot be a potential function such that the amortized cost of Insert is O(1), and the amortized cost of ExtractMin is O(log n).

3 Problem 2a

Show how to implement a queue with two ordinary stacks so that the amortized cost of each ENQUEUE and each DEQUEUE operation is $O(1)$
Consider 2 stacks, in and out. Let us define the following operations to follow these steps of instructions:

- ENQUEUE: Push the new element onto in
- DEQUEUE:
  - If out is empty, refill out by popping each single element from the stack in onto out
  - Else, pop and return the top element from outbox
- MULTI-DEQUEUE(Q,k): Dequeue $min(|Q|, k)$ elements from Q

To show that the amortized cost of all the above 3 operations are $O(1)$:
Consider the following analysis using the accounting method where we pay 4 credits for every push of an element. 1 of these credits is the actual cost of the push, the second one is saved for when the element is popped from in, the third is saved for when the element is pushed onto out (the second and the third capture the transfer operations) and lastly, we spend the fourth credit when the element is popped from out. Since we have credits saved up for all operations besides the initial push onto in, all of the other operations have cost 0.
Thus, when we consider a sequence of n operations, each of them can at worst have a cost of 4. The aggregate cost is 4n= $O(n)$ for n operations. Thusly, amortized cost of each operation is $O(1)$. We should also note that this amortized cost is always an upper bound for the actual cost since the credit count is always non negative.

4 Problem 2b

It is not possible to reverse the roles of stacks and queues in the above problem due to the prime reason that when we dequeue from the queue in, and enqueue to the queue out, that is - when we perform the transfer operation; the queue does not change its ordering at all. Thus, we cannot make use of the property of a stack that gets reversed when upon the transfer operation to cut down on the number of times we need to transfer (an expensive operation). So, every time we need to dequeue, we will need to transfer $n-1$ elements from the stack in use to the other stack and eventually return the last remaining element. This is an expensive $O(n)$ operation at every step and there is no way to distribute one transfer operation to take case of multiple calls to dequeue since the queue is never reversed. Thus, each and every dequeue will be expensive and we can never have an implementation using two queues to mimic a stack and still have $O(1)$ amortized time complexity for operations PUSH AND POP. Note that if we did not need $O(1)$ time complexity, we could have easily solved this by transferring the queue from the active one to the non-active one and returning the last element of the queue at every step resulting in costs of $O(n)$ at every step, even under amortized analysis.

5 Problem 3

Design a data structure to support the following two operations for a dynamic multiset S of integers, which allows duplicate values:
INSERT(S, x) inserts x into S. DELETELARGER(S, k), which deletes the largest ceiling $(|S|/k)$ elements from S for any given $1 <= k <= 4$.
Explain how to implement this data structure so that any sequence of m INSERT and DELETELARGER operations runs in O(m) time. Your implementation should also include a way to output the elements of S in $O(|S|)$ time.
Consider a list L. Define the operations as follows: INSERT(S,x) - append x to the end of the array DELETE-LARGER(S,x)- A modified version of quicksort as follows

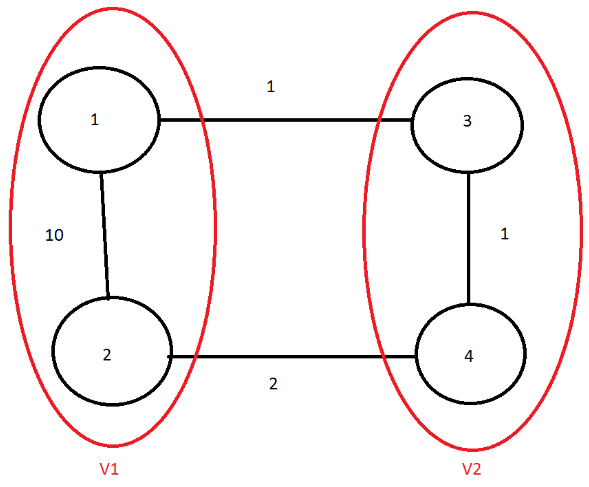
```
DELETE-LARGER(S,k):
    Compute y=$\lceil |S|/k \rceil$ where $1 <= k <= 4$
    pick a pivot randomly and count the number of elements larger
    than it = a
    reorder elements s.t. all the elements less than a come before
    the pivot and all after a will be elements larger than a.
    #this reordering takes $O(n)$ time
    Record the number of variables after the pivot as i
    Case1: $y = i$
        Simply remove these elements that are greater than the
    pivot in O(y) time
    Case2: $y < i$
        Recursively call DELETE-LARGER(S2,k) where S2 is the set of
     elements larger than the pivot
    Case3: $y > i$
        Delete the i elements from S
        update $k'$ such that the y calculated with this new $k'$ is
    equal to the original $y-i$
        and call DELETE-Larger(S,k')
    #this will be O(n) time when calculated using the master
    theorem analysis
```

Alternatively, we could use the divide and conquer approach we saw in class to find the $\frac{n}{k}$th element. This takes $O(n)$ time. We use this element as our pivot and now go through the L to delete all elements larger than it. This will take $O(n)$ time as well. Thus, total run time is $O(2n) = O(n)$ time.

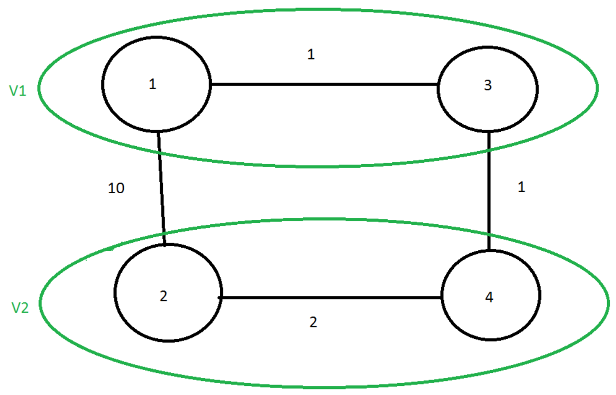
Firstly, let's note that we can easily output every element in our list L in $O(|S|)$ time. Also note that INSERT will have O(1) run time and DELETE-LARGER will have $O(n)$ time by seeing that the individual steps are $O(n)$ in sequence and the $k = 3$ step can be shown to be $O(n)$ where n is the number of elements we have using master theorem analysis. Having define our two operations, let us now prove that a sequence of m of these operations take $O(m)$ time.
Consider $\phi(S_i) = 2a|L|$:
INSERT(S,x): The actual cost $c_i = 1$.
And amortized cost will be, $\hat{c_i} = c_i + \phi(S_i) - \phi(S_{i-1}) = 1 + 2a|L| - 2a(|L|-1) = 1 + 2a = O(1)$
DELETE-LARGER-HALF(S): The actual cost $c_i = am$.
And amortized cost will be, $\hat{c_i} = c_i + \phi(S_i) - \phi(S_{i-1}) = a|L| + 2a\frac{\lceil |L| \rceil}{k} - 2a|L|$.
Since we can just set the constants to be $0.5k = 0$
Therefore the amortized times are $O(1)$ for both operations and the m operations run in $O(m)$ time.

6 Problem 4

The algorithm is incorrect depending on the choice of our set partitions V_1 and V_2 of the original set V. It won't work if we choose vertex sets such that V_1 or V_2 contain an edge that will be chosen in the MST of V_1 but not in the MST of the V. Consider the following counterexample:



Prof. Borden's algorithm will first look at the individual partitions V_1 and V_2 and select the minimum weight edges connecting the two nodes in each set by using the recursive procedure detailed in the problem. It'll thus select the edge from 1 to 2 in V_1 and the edge from 3 to 4 in V_2. Now, it'll look for the minimum edge connecting these two partitioned sets- it'll choose the lower weight edge between the edge from 1 to 3 and 2 to 4- it'll choose the edge from 1 to 3 with weight 1. Thus, our MST selected is composed of the following edges; 1 to 2, 1 to 3 and 3 to 4. However, this is not an MST for our original graph. The actual MST is composed of the edges from 1 to 3, 3 to 4 and 4 to 2. Thus, We can see that the proposed algorithm is not perfect when partitioning is implemented incorrectly.
Note: Had we used partitioning as in the below image for the same graph, we would have resulted with the correct MST using Prof. Borden's algorithm.



We would pick the actual MST; composed of the edges from 1 to 3, 3 to 4 and 4 to 2.

7 Problem 5

Notation: $W_{1,2}$ = weight of the edge between nodes 1 and 2 in our graph. $v.d$ is the shortest distance path to a vertex from a source node as produced by the professor's algorithm. $v.\pi$ is the predecessor node on the shortest path to the vertex v from the source node as produced the algorithm.

```
for each edge in edgeset:
    $u = v.\pi$
    #the vertex on the other end of the edge to our
    #destination vertex
    check that $u.d + W_{u,v} >= v.d$
#this is $O(|E|)$ time
for each vertex in vertexset:
    if vertex==source:
        check that $vertex.\pi = None$ and $vertex.d = 0$
    elif $vertex.\pi = None$:
        check that $vertex.d = \infty$
    else:
        check that $vertex.\pi! = vertex$
        check that $vertex.\pi.d + W_{(v.\pi),v} = v.d$
#this will take $O(|V|)$ time
```

Thus we have two procedures in sequence that that $O(|E|)$ and $O(|V|)$ time respectively. So we can check the professor's algorithm in $O(|V + E|)$ time. We have a $O(|V + E|)$ algorithm to check the output of Prof. Gaedel's algorithm!