NORTHWESTERN UNIVERSITY

EECS 336 – Homework 6

Weihan Chu

May 19, 2016

1 QUESTION 1

1.1 PART A

Let d_i is the depth in the binary min-heap of element i. We define the potential function Φ of heap H as follows:

$$\begin{aligned}\Phi(H_i) &= \sum_{k=1}^{i} d_k \\ &= \sum_{k=1}^{i} lgk\end{aligned}$$

From above, we can easily get that for any $i > 0$, there has:

$$\Phi(H_i) > \Phi(H_0) = 0$$

So the total amortized cost of a sequence of n INSERT and EXTRACT-MIN operations is an upper bound on the total actual cost.// For the INSERT operation, we first insert the new element in the end of the heap with O(1) time then bubble up the element with at most $logn_i$ time. So the total time complexity of the insert is $logn_i$. Then because we insert a new node, the Φ increased $d_i = logn_i$.Now we can get:

$$\begin{aligned}c_i^1 &= c_i + \Phi(H_i) - \Phi(H_i - 1) \\ &= logn_i + logn_i \\ &= 2logn_i \\ &= O(logn)\end{aligned}$$

Then for the EXTRACT-MIN operation, we first remove the least element from the root with O(1) time and the bubble up the elements with $log(n_i)$ time. Since the top element is removed from the heap. And other elements bubble up. Seems like the last element is removed. So the Φ deceased $O(logn_i)$. So we can get that:

$$\begin{aligned} c_i^1 &= c_i + \Phi(H_i) - \Phi(H_i - 1) \\ &= logn_i - logn_i \\ &= O(1) \end{aligned}$$

1.2 PART B

I think the answer is no. Since $c_i^1 = c_i + \Phi(H_i) - \Phi(H_i - 1)$ and $c_i = O(logn)$ in both INSERT and EXTRACT-MIN. If we want to make INSERT is O(1) and EXTRACT-MIN is $O(logn)$. We need to make the $\Phi(H_i) - \Phi(H_i - 1)$ in INSERT to be negative and the $\Phi(H_i) - \Phi(H_i - 1)$ in EXTRACT-MIN to be positive. Since it's not possible for us to find such function. So the answer for this question is no.

2 QUESTION 2

2.1 PART A

We use two stacks A and B to implement queue.

```
        ENQUEUE: Push n into A
        DEQUEUE: if B is not empty
                    POP B
                 else if B is empty
                    Do
                        POP A
                        PUSH B
                    until A is empty
                    POP B
MULTI-DEQUEUE: if B is not empty
                    POP B multiple times
                    if there is no enough elements
                      POP from A to get enough elements
                 else if B is empty
                     Do
                       POP A
                       PUSH B
```

```
until A is empty
POP B  multiple times
```

Let n_a is the number of elements in A.If we define the potential function Φ as follows: $\Phi(H_i) = 2n_a$,we can get $\Phi(H_i) > \Phi(H_0) = 0$.
For ENQUEUE, the time complexity is O(1) and the potential function is $(2n - 2(n-1)) = 1$.Now we can get:

$$\begin{aligned} c_i^1 &= c_i + \Phi(H_i) - \Phi(H_i - 1) \\ &= 1 + 1 \\ &= O(1) \end{aligned}$$

For the DEQUEUE, if B is not empty,we just need to pop one element from B. So the time complexity is O(1) and the number of elements in A doesn't change. So we can get:

$$\begin{aligned} c_i^1 &= c_i + \Phi(H_i) - \Phi(H_i - 1) \\ &= 1 + 0 \\ &= O(1) \end{aligned}$$

if B is empty,we need to first POP all elements form A and push all elements into B and then POP one element from B. So the time complexity is $2n_a+1$. For the potential difference:$\Phi(H_i) - \Phi(H_i - 1) = 0 - 2n_a = -2n_a$, so we can get:

$$\begin{aligned} c_i^1 &= c_i + \Phi(H_i) - \Phi(H_i - 1) \\ &= (2n_a + 1) - (2n_a) \\ &= O(1) \end{aligned}$$

For the Multi-DEQUEUE, the situation is same, we can get that :

$$\begin{aligned} c_i^1 &= c_i + \Phi(H_i) - \Phi(H_i - 1) \\ &= O(1) \end{aligned}$$

In this way,we can get O(1) for every operation.

2.2 PART B

I think the answer of this question is no. We use two queues to implement stack. When we push elements, we just choose a queue which is not empty(whichever when both are empty). When we do pop, first pop all elements of the queue except the tail into another empty queue, and then pop the tail which is your want. When we do multi-pop, for example,we want to pop n elements from the queue. We just need to pop size-n elements into another queue. And then pop the last n elements.

3 QUESTION 3

INSERT take O(1) because we insert into a unsorted array.For DELETE-LARGER(s,k), we can use divide and conquer to solve ,which will take O(n) to find the pivot and partition the array and delete the larger in O(1).
We give m to INSERT amortized cost. When we insert an element, 1 of m is used for the insert cost and the other m-1 is stored for future use. When we do the DELETE-LARGER(s,k) function, we take 1 from every element to do the find pivot, which is the $[(|s|/k)]$th largest element. So after this , ever element now has m-2. Then, the largest $[(|s|/k)]$ use 1 to delete. So, at this moment, we can get:

$$[(|s|/k)](m-3) = |S| - [(|s|/k)]$$

$$m = k+2$$

So, the insert need $k+2$ and DELETE-LARGER(s,k) need 0. And the two steps' amortized cost is O(1). So we can get the final result that this operation has O(m) time complexity.

4 QUESTION 4

I think this algorithm is not right. I draw a example picture named picture(problem4).pdf. So you can open my pdf to see this simple example. If we partition G into V1 and V2 as :

$$V1 = [A, B], V2 = [C, D]$$

Then we follow the algorithm, we find that there are two edges across the trees(A, C) and (B, D), we also find that (A, C) is smaller than(B, D). So we add (A, C) into our tree. So by follow this algorithm, we get the final tree is $[(A, B), (C, D), (A, C)]$. But actually, the real result is$[(A, B), (A, C), (B, D)]$. So this algorithm get the wrong result. This algorithm is not right.

5 QUESTION 5

We need to do following steps to check the correctness of the algorithm:
a. We first check every edge in this graph. For each edge u to v. We check whether they satisfied this condition:

$$u.d + w_{(u,v)} \geq v.d$$

we need $O(E)$ time to do this check.
b.We then check every vertex v. For the source vertex, we check that the source(s) vertex's parent is null, which means $s.\pi = null$. Then we check other vertex. They should satisfy this condition:

$$v.\pi.d + w_{(v.\pi,v)} = v.d$$

which means the distance of every vertex' parent plus the distance from parent to this vertex equals the distance of this vertex. We check this in $O(V)$
c. $s.d = 0$, which means s is the source vertex. We can check this property in $O(1)$
d.At last, we run DFS on the tree from source. Every vertex should be visited,which means this result can represent a spanning tree. Since there are n-1 edges. So this step takes $O(E)$
So the total time complexity is $O(V + E)$. Since the situation is that if the result doesn't satisfy any condition above , the result is wrong. If the result satisfy all condition above , the result is right. So this prove algorithm is right.