Problem Set for Advanced Algorithms - Part 1

(Corman, Leiserson, Rivest, Stein “Introduction to Algorithms”)

17.1-1 If the set of stack operations included a MULTIPUSH operation, which pushes k items onto the stack, would the $O(1)$ bound on the amortized cost of stack operations continue hold?

Solution:

No, because there is no limits in number of MULTIPUSH operations like it is in MULTIPOP. ■

17.1-2 Show that if a DECREMENT operation were included in the k-bit counter example, n operations could cost as much as $\Theta(nk)$ time.

Solution:

Start point: all k bits are set to zero: $00...00$.

Perform INCREMENT until the last bit is set to one and all the others are set to zero: $100...00$.

At this point a DECREMENT operation would cost k, all the zeros must turn into one and the one must turn into zero: 011...11.

Then if we perform an INCREMENT operation, it would cost k again, all ones turn into zero and the zero turn into one: 100...00.

By repeating this DECREMENT and INCREMENT operations n times, the total amount of iterations is $O(nk)$. ■

17.2-1 Suppose we perform a sequence of stack operations on a stack whose size never exceeds k. After every k operations, we make a copy of the entire stack for backup purposes. Show that the cost of n stack operations, including copying the stack, is $O(n)$ by assigning suitable amortized costs to the various stack operations.

By assigning 2 to PUSH operation and zero to the others, it is sufficient not considering the backup. So if we assign 3 to PUSH operations and one to the others, then for sure after k operations there are k extra in the bank that can be used to backup. Since the size of the stack never exceeds k, the extra k in the bank is sufficient to perform the backup. ■

17.3-3 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 s.t. the amortized cost of INSERT is $O(\lg n)$ and the amortized cost of EXTRACT-MIN is $O(1)$, and show that it works.

Solution:

Remember in the potential method we have $\hat{c_i} = c_i + \Phi(D_i) - \Phi(D_{i-1})$.

$$\Phi(D_i) = \sum_{elem \in D_i} elem.layer.$$

- If i^{th} operation is INSERT:

  $c_i = D_{i-1}.depth$

  CASE 1: The heap is full before INSERT.

  $\Phi(D_i) = \Phi(D_{i-1}) + D_{i-1}.depth + 1.$

  $\hat{c_i} = D_{i-1}.depth + \Phi(D_{i-1}) + D_{i-1}.depth + 1 - \Phi(D_{i-1}) = 2 * D_{i-1}.depth + 1.$

  CASE 2: Otherwise.

$\Phi(D_i) = \Phi(D_{i-1}) + D_{i-1}.depth.$

$\hat{c_i} = D_{i-1}.depth + \Phi(D_{i-1}) + D_{i-1}.depth - \Phi(D_{i-1}) = 2 * D_{i-1}.depth.$

In any case, the amortized cost is $O(\lg n)$.

- If i^{th} operation is EXTRACT-MIN:

  $c_i = D_{i-1}.depth.$

  $\Phi(D_i) = \Phi(D_{i-1}) - D_{i-1}.depth.$

  $\hat{c_i} = D_{i-1}.depth + \Phi(D_{i-1}) - D_{i-1}.depth - \Phi(D_{i-1}) = 0.$

  The amortized cost is O(1). ■

17.3-5 Suppose that a counter begins at a number with b 1s in its binary representation, rather than at 0. Show that the cost of performing n INCREMENT operations is $O(n)$ if $n = \Omega(b)$. (Do not assume that b is constant.)

Solution:

Let $\Phi(D_0) = b_0$ and $\Phi(D_n) = b_n$.

The amortized coast to a INCREMENT is 2, then

$$\sum_{i=1}^{n} \hat{c_i} = \sum_{i=1}^{n} (c_i + \Phi(D_i) - \Phi(D_{i-1})) = \sum_{i=1}^{n} 2 - \Phi(D_0) + \Phi(D_n) = 2n - b_0 + b_n.$$

Since $n = \Omega(b)$ and b_1, b_n and n are positive numbers, we have the following:

$$n \geq b_n \geq b_n - b_1 \geq -b_1 \geq -n$$

$$\Rightarrow 3n \geq 2n + b_n - b_1 \geq n.$$

Then performing n INCREMENT operations is $O(n)$. ■

17.3-6 Show how to implement a queue with two ordinary stacks so that the amortized cost of each ENQUEUE and each DEQUEUE operation is $O(1)$.

Solution:

Let the two stacks being S_1 and S_2.

Define the ENQUEUE operation to be a PUSH operation in S_1.

Separate DEQUEUE operation in 2 cases:

CASE 1: S_2 is not empty.

Then define the DEQUEUE operation to be a PULL in S_2.

CASE 2: S_2 is empty.

Then do a PULL operation in S_1 and PUSH the returned element into S_2. Repeat this until S_1 is empty. Finally, do a PULL operation in S_2 and return the result.

Clearly this algorithm works correctly.

Now let's prove these operations has amortized cost $O(1)$.

Assign the costs being 3 to ENQUEUE and 1 to DEQUEUE.

When we do an ENQUEUE operation we put 2 in the bank.

When we do a DEQUEUE operation, we have two cases:

CASE 1: if S_2 is not empty we put nothing in the bank but also we do not take nothing from the bank.

CASE 2: if S_2 is empty we need to perform PULL and PUSH operations until S_1 is empty, which will cost $2|S_1|$. Since for each element we pushed into S_1 we put 2 in the bank, we have for sure $2|S_1|$ in the bank. After that we need to perform a PULL operation into S_2, then use the cost 1 of DEQUEUE to perform this operation.

Conclusion: the algorithm works and the amortized cost for each operation is $O(1)$. ■

19-1 (Alternative implementation of deletion) Professor Pisano has proposed the following variant of the FIB-HEAP-DELETE procedure, claiming that it runs faster when the node being deleted is not the node pointed to by $H.min$.

```
PISANO-DELETE($H, x$)
1  if $x == H.min$
2     FIB-HEAP-EXTRACT-MIN($H$)
3  else $y = x.p$
4     if $y \neq$ NIL
5        CUT($H, x, y$)
6        CASCADING-CUT($H, y$)
7     add $x$'s child to the root list of $H$
8     remove $x$ from the root list of $H$
```

a. The professor's claim that this procedure runs faster is based on the assumption that line 7 can perform in $O(1)$ actual time. What is wrong with this assumption?

b. Give a good upper bound on the actual time of PISANO-DELETE when x is not $H.min$. Your bound should be in therms of $x.degree$ and the number c of calls to the CASCADING-CUT procedure.

c. Suppose that we call PISANO-DELETE(H, x), and let H' be the Fibonacci heap that results. Assuming that node x is not a root, bound the potential of H' in terms of $x.degree$, c, $T(H)$, and $m(H)$.

d. Conclude that the amortized time for PISANO-DELETE is asymptotically no better than for FIB-HEAP-DELETE, even when $x \neq H.min$.

Solution:

a. In fact it is bounded by the degree of x, since for each child of x we need to update it's parent to be null. □

b. $O(c + x.degree)$. □

c. Since we add each child of x as a new root in H then $T(H') = T(H) + x.degree$.

If $m(H)$ increases, it can only increase by one, then $m(H') \leq m(H) + 1$.

In conclusion we have that

$$\Phi(H') = T(H') + 2m(H') = T(H) + x.degree + 2m(H') \leq T(H) + x.degree + 2(m(H) + 1). \quad \square$$

d. Phi(H') is $O(x.degree) = O(\ln n)$, same as FIB-HEAP-DELETE. ■

26.1-7 (More Fibonacci-heap operations) We wish to augment a Fibonacci heap H to support two new operations without changing the amortized running time of any other Fibonacci-heap operations.

a. The operation FIB-HEAP-CHANGE-KEY(H, x, k) changes the key of node x to the value k. Give an efficient implementation of FIB-HEAP-CHANGE-KEY, and analyze the amortized running time of your implementation for the cases in which k is grater than, less than, or equal to $x.key$.

b. Give an efficient implementation of FIB-HEAP-PRUNE(H, r), which deletes $q = min(r, H.n)$ nodes from H. You may choose any q nodes to delete. Analyze the amortized running time of your implementation. (Hint: You may need to modify the data structure and potential function.)

Solution:

a. Algorithm:

```
FIB-HEAP-CHANGE-KEY$(H, x, k)$
1   if $x.key > k$
2       FIB-HEAP-DECREASE-KEY$(H, x, k)$
3   else if $x.key < k$
4       Let $y$ a new node with y.key = k
4       FIB-HEAP-INSERT$(H, y)$
5       FIB-HEAP-DELETE$(H, x)$
```

If $k < x.key$ the amortized complexity is equal the complexity of perform FIB-HEAP-DECREASE-KEY, then $O(1)$.

If $k > x.key$ then the amortized complexity is equal the complexity of perform FIB-HEAP-INSERT and FIB-HEAP-DELETE, then $O(1) + O(\lg n) = O(\lg n)$.

If $k = x.key$, we do nothing, then the complexity is $O(1)$. □

b. Create a list of leafs and change the operations to update it when necessary.

Change the potential function to be $\Phi(H) = T(H) + 2m(H) + |H|$, where |H| is the size of the structure.

|H| can only increase during FIB-HEAP-INSERT operation and only by 2, one for the heap function and one for the list of leaves. Then it will not change the amortized cost for any operation.

Algorithm:

```
FIB-HEAP-PRUNE$(H, r)$
1   $r = min(r, H.n)$
2   while r > 0
3       $x = H.leaves.pop()$
4       let $y = x.parent$
5       remove $x$ from $H$
6       if $x$ was the only child of $y$
7           $H.leaves.push(y)$
```

After n operations, the sum of potential function for the FIB-HEAP-INSERT operations will be always grater than the sum of potential functions for the FIB-HEAP-PRUNE operations. In other words, the algorithm is $O(H.n)$, but since we can only delete nodes that were inserted, then the amortized complexity is O(1). ■

26.1-7 Suppose that , in addition to edge capacities, a flow network has **vertex capacities**. That is each vertex v has a limit $l(v)$ on how much flow can pass through v. Show how to transform a flow network $G = (V, E)$ with vertex capacities into an equivalent flow network $G' = (V', E')$ without vertex capacities, such that a maximum flow in G' has the same value as a maximum flow in G. How many vertices and edges does G' have?

Solution:

Transform each vertex into an edge s.t. it's incoming edges are connected to one end of the edge and it's outgoing edges are connected to the other end. Make the weight of these edges to be equal the weight of the respective vertex.

More formally, for each vertex v_i with weight w_{v_i}, create an edge $e_{v_i} = (x_{v_i}, y_{v_i})$ with weight w_{v_i}. For each v_i incoming edge, say (v_j, v_i), let it be (v_j, x_{v_i}) instead, and for each v_i outgoing edge, say (v_i, v_j), let it be (y_{v_i}, v_j) instead.

Since each vertex is turning into an edge, for each vertex of G we have one more edge and one more vertex in G'. Then $|V'| = 2|V|$ and $|E'| = |E| + |V|$. ■

26.2-12 Suppose that you are given a flow network G, and G has edges entering the source s. Let f be a flow in G in which one of the edges (v, s) entering the source has $f(v, s) = 1$. Prove that there must exist another flow f' with $f'(v, s) = 0$ s.t. $|f| = |f'|$. Give an $O(E)$-time algorithm to compute f', given f, and assuming that all edge capacities are integers.

Solution:

Let e being an s incoming edge (v, s) s.t. $f(v, s) = 1$. Since s is the source, it cannot have incoming flow grater than outgoing flow, than there must exist an edge e' outgoing s s.t. it's flow is grater or equal than 1.

Moreover, there must be a cycle C passing though e and e' s.t. all the edges in the cycle has a flow grater or equal than 1. Otherwise, there will be a vertex different than s and t s.t. incoming and outgoing flow is different, contradicting the conservation flow rule.

Finally, if we reduce the flow by 1 in the entering cycle, it will not affect the global value of the flow, and then $|f| = |f'|$.
□

Now, we construct an algorithm to find f', given the flow network G, flow f and

```
NEW-FLOW($G, f$)
1   let $s$ the source of G
2   let $(v, s)$ a $s$ incoming edge with weight $= 1$
3   find $P$ a path from $s$ to $v$ in $G$ s.t. for all edges $e$, $f(e) > 0$
5   let $f' = f$
6   **for** $e$ **in** $E(P) \cup \{(v, s)\}$
7       $f'(e) = f(e) - 1$
8   **return** f'
```

The complexity of this algorithm is the sum of complexities to find the path P and then change the flow of each edge of P, then $O(V + E) + O(E) = O(V + E)$. Since the flow network is connected, $V < 2E$. Then the complexity of the algorithm is $O(E)$. ■