EECS 336

Litong Wang

Homework 6

May 18, 2016

Problem 1

Solution:

Define configuration D_i the heap after the ith operation, and consists of n_i elements. d_j is the depth of element j in the correspoding binary tree. We can define the potential function $\Phi(D_i)$ as follow:

$$\Phi(D_i) = \begin{cases} 0 & \text{if } i = 0 \\ \sum_{j=0}^{n_i} d_j & \text{if } i > 0 \end{cases} \tag{1}$$

If we start with an empty heap, then we have $\Phi(D_0) = 0$, and we always maintain that $\Phi(D_i) \geq 0$.

If the ith operation is Insert, then $n_i = n_{i-1} + 1$. If we start with an empty heap, $n_{i-1} = 0$ and $n_i = 1$. And the amortized cost is:

$$\begin{aligned} \widehat{c_i} &= c_i + \Phi(D_i) - \Phi(D_{i-1}) \\ &= \log 1 + 1 - 0 \\ &= 1 \end{aligned}$$

If we insert the element into an nonempty, we attach the element to the leaf of the binary tree and do *heapup*. Then $n_i = n_{i-1} + 1$, $\Phi(D_i) - \Phi(D_{i-1}) = \log n_i$, the amortized cost is:

$$\begin{aligned} \widehat{c_i} &= c_i + \Phi(D_i) - \Phi(D_{i-1}) \\ &= \log n_i + \log n_i \\ &= 2 \log n_i \\ &= \mathcal{O}(\log n) \end{aligned}$$

If the ith operation is Extract-Min, then $n_i = n_{i-1} - 1$. If the ith operation extracts from a heap with only one element, then $n_i = 0$ and $n_{i-1} = 1$. And the amortized cost is:

$$\begin{aligned} \widehat{c_i} &= c_i + \Phi(D_i) - \Phi(D_{i-1}) \\ &\leq k \ln 1 + 0 - k \cdot 1 \ln 1 \\ &= 0 \end{aligned}$$

If the ith operation is Extract-Min and extracts from a heap with more than one element, then $n_i = n_{i-1} - 1$, $\Phi(D_i) - \Phi(D_{i-1}) = -\log n_{i-1}$ and $n_{i-1} \geq 2$, and the amortized cost is:

$$\begin{aligned}
\widehat{c_i} &= c_i + \Phi(D_i) - \Phi(D_{i-1}) \\
&= \log n_i - \log n_{i-1} \\
&= \log n_i - \log(n_i + 1) \\
&\leq 1 \\
&= \mathcal{O}(1)
\end{aligned}$$

b) No, we cannot find a potential function such that the amortized cost of Insert is $\mathcal{O}(1)$ and the amortized cost of Extract-Min is $\mathcal{O}(\log n)$.
In worst case, we need to take $\Omega(\log n)$ time to do Insert operation. The actual cost of Insert operation in worst case the $\Theta(\log n)$. Because the actual cost of a Insert operation is $\mathcal{O}(\lg n)$. Assume we come up with a potential function such that the amortized cost of Insert is $\mathcal{O}(1)$, which means:

$$\begin{aligned}
\widehat{c_i} &= c_i + \Phi(D_i) - \Phi(D_{i-1}) \\
&= \log n + \Phi(D_i) - \Phi(D_{i-1}) \\
&= k^* (k^* \ \textit{is a constant})
\end{aligned}$$

Hence, this means $\Phi(D_i) - \Phi(D_{i-1}) < 0$. The potential difference of the ith operation is negative, the amortized cost represents an undercharge to the ith operation, and the decrease in the potential pays for the actual cost of the operation. This means our amortized cost of Insert is lower than the actual cost. This will cause a problem. If we only do n Insert operation, then $\sum_{i=0}^{n} \widehat{c_i} < \sum_{i=0}^{n} c_i$. And if we don't do Extract-Min, then we cannot guarantee that the amortized cost is larger than the actual cost. This is different from problem a, because although we undercharge the Extract-Min operation, we overcharge the Insert operation to pay for it. And if we want to do Extract-Min operation, we must do Insert operation first. This guarantees that the amortized cost is larger than the actual cost. In conclusion, we can overcharge Insert operation and undercharge Extract-Min operation, but we cannot undercharge Insert operation and overcharge Extract-Min operation. Hence, we cannot find a potential function such that the amortized cost of Insert is $\mathcal{O}(1)$ and the amortized cost of Extract-Min is $\mathcal{O}(\log n)$.

Problem 2

Solution:

a) Let's say we have two stacks A and B. We use stack A to do Enqueue operation and use stack B to do Dequeue operation. In case of a dequeue operation, if stack B is not empty, we can just use a Pop operation to do Dequeue operation. If stack B is empty, we need to pop all the elements from stack A and push it into stack B, then we can do the dequeue operation.
The actual costs of operations were:

$$\begin{aligned} Enqueue &\quad 1, \\ Dequeue &\quad s, \\ Multi-Dequeue &\quad 2\min(k, s), \end{aligned}$$

where k is the arguement supplied by Multi-Dequeue and s is the size of stack A. We can assgin the following amortized costs:

$$\begin{aligned} Enqueue &\quad 4, \\ Dequeue &\quad 0, \\ Multi-Dequeue &\quad 0, \end{aligned}$$

We can pay for any sequence of queue operations by charging the amortized costs. Suppose we use a dollar bill to represent each unit of cost. We start with two empty stacks A and B. In case of Enqueue operation, we push one element into stack A, and we use 1 dollar to pay the actual cost of the Push and are left with a credit of 2 dollar. At any point in time, every element on the stack has 2 dollar of credit on it.
The dollar stored as credits serves as prepayment for the future cost. One dollar of popping it from the stack A, one dollar of pushing it into stack B and one dollar of popping it from stack B. When we execute a Dequeue operation, we charge the operation nothing and pay its actual cost using the credit stored in the stack. To dequeue an element from our queue, if stack B is not empty, we take one dollar of credit from the bank and use it to pay the actual cost of the operation. If stack B is empty, we need to pop all elements from stack A, we take one dollar of credit from the bank at each poping. And we push all these elements into stack B, we take one dollar of credit from the bank to pay for the actual cost. And we use one dollar of credit from the bank to pay for the actual cost of Dequeue which actually is poping from stack B. Thus, by charging the Enqueue operation a little bit more, we can charge the Dequeue operation nothing.
Moreover, we can also charge Multi-Dequeue operations nothing. If stack B is not empty, to Dequeue the first element, we take the dollar of credit from the bank and use it to pay the actual cost of a Pop operation. To Dequeue a second element, we again have a dollar

of credit from the bank to pay for the Pop operation, and so on. If stack B is empty, we Pop all elements from stack A and Push it into stack B, we can use credits to pay for the actual costs. And then we Pop elements from stack B to finish the Multi-Dequeue operation. Thus, we have always charged enough up front to pay for Multi-Dequeue operations. In other words, since each element on the stack A has 3 dollar of credit on it, and the stack always has a nonnegative number of elements, we have ensured that the amount of credit is always nonnegative. Thus, for any sequence of n Enqueue, Dequeue, and Multi-Dequeue operations, the total amortized cost is an upper bound on the total actual cost. Since the amortized cost of Enqueue, Dequeue and Multi-Dequeue are all constant time, so their amortized cost are all $\mathcal{O}(1)$.

b) Let's say we have two queues A and B. At first, both queues are empty, if we want to do a Push operation, we can choose one of the two queues to do a Enqueue operation. If none of the two queues are empty and it is a Push operation, we need to Enqueue the element into the empty queue, then we Dequeue all the elements from the non-empty queue, and Enqueue all these elements into the other queue. In this way, elements which are Push earlier will at the end of the queue. If it is a Pop operation, we can just simply do a Dequeue operation to the non-empty queue. By this method, we can use two queue to implement a stack.

The amortized cost of each operation cannot be $\mathcal{O}(1)$, because if we insert n elements, we need to transfer the elements between these two queues $n-1$ times. And we need to transfer $1, 2, \cdots, n-1$ elements. That will require $\frac{n(n-1)}{2}$ operations, if we use accouting method, we need to pay $\frac{n-1}{2}+1$ dollar during each Push operation. Clearly, it is not $\mathcal{O}(1)$, hence, we cannot use two queues to implement a stack that allows the amortized cost of each Enqueue, each Dequeue and each Multi-Dequeue operation to be $\mathcal{O}(1)$.

Problem 3

Solution:

We could use dynamic tables which supports Insert, Delete and can find the kth element in $\Theta(n)$ time(use the algorithm for finding the kth element we discussed in class) to solve this problem. We define the load factor $\alpha(T)$ of a nonempty table to be the number of items stored in the table divided by the size (number of slots) of the table. The load factor of an empty table is 1. If the load factor is 1, and we need to expand this table. If the load factor is smaller than $\frac{1}{4}$, we need to do contraction operation. The amortized cost of Insert and Delete operation are $\mathcal{O}(1)$ as discussed in the textbook. If it is an Insert operation, we just attach the element at the end of the table. If it is a Delete-Larger(S,k) operation, we first use QuickSelection algorithm to select the $(\frac{S}{k})$th element and do partition. Then we do Delete operations to largest $(\frac{S}{k})$ elements.

We can assign the following amortized costs:

$$\begin{aligned} Insert &\quad 10, \\ Delete - Larger(S,k) &\quad 0 \end{aligned}$$

Here is the assginment of each dollar:

$$\begin{aligned} Insert &\quad 1, \\ Expansion &\quad 2, \\ Contaction &\quad 2, \\ Delete - Larger(S,k) &\quad 5 \end{aligned}$$

We can pay for any sequence of Insert and Delete-Larger(S,k) by charging the amortized costs. Suppose we use a dollar bill to represent each unit of cost. We start with an empty table. In case of Insert operation, we insert one element into table, and we use 1 dollar to pay the actual cost of the Insert and are left with 8 dollar. If an Insert operation causes an expansion, we use 1 dollar to pay for moving itself and another 1 to pay for moving another element. If a Delete-Larger(S,k) operation causes a contraction, then we use one 1 to pay for removing itself and another 1 to pay for removing another element. At any time, every element in the table has a credit with 5 dollar.
In case of a Delete-Larger(S,k) operation, The key idea is that we use the credits left by the element deleted last time to pay for the next Delete-Larger(S,k) operation. We charge every element 1 dollar for Search for the $(\frac{S}{k})$th element and do partition. Then we remove these elements. In case of $k = 4$, we remove $\frac{S}{4}$ elements, deleting each element costs 1 dollar. Then, we assign the credits these elements hold to the remaining $\frac{3S}{4}$ elements. In this way, we can guarantee that every element in the table has a credit with at least 5 dollar. If $k < 4$, the number of elements we removed is larger than $\frac{S}{4}$, and we assign the credits these elements hold to the remaining $\frac{(k-1)S}{k}$ elements. Than each remaining element will have credits larger than 5 dollar. Because each element in table has credit at least 5 dollar, we can use this credit to pay for another Delete-Larger(S,k) operation. Because the amortized cost of Insert and Delete Larger operation are all constant, so any sequence of m Insert and Delete-Larger(S,k) operations runs in $\mathcal{O}(m)$ time.

Problem 4

Solution:

This algorithm is incorrect. We can easily find a counter example for it. Assume we have six vertiex a, b, c, d, e, f and we have seven edges $(a, b), (a, c), (c, e), (b, d), (c, d), (d, f), (e, f)$.

The weights of these edges are $w(a,b) = 1, w(a,c) = 100, w(c,e) = 100, w(b,d) = 100, w(c,d) = 2, w(d,f) = 100, w(e,f) = 3$. Acorrding to the algorithm, if we let $V_1 = \{a,c,e\}, E_1 = \{(a,c),(c,e)\}$ and $V_2 = \{b,d,f\}, E_2 = \{(b,d),(d,f)\}$. And we can construct two minimum spanning trees T_1, T_2 from $G_1 = (V_1, E_1), G_2 = (V_2, E_2)$. And we select the minimum-weight edge in E that crosses the cut (V_1, V_2) which is (a,b) to connect T_1 and T_2. We can get a spanning tree T. The total weight of this tree is $w(a,c) + w(c,e) + w(a,b) + w(b,d) + w(d,f) = 401$. However, we can easily find out that the we should connect edges $(a,b),(b,d),(c,d),(c,e),(e,f)$ to construct the minimum spanning tree. The total weight of this tree is $w(a,b) + w(b,d) + w(c,d) + w(c,e) + w(e,f) = 206$.
This algorithm is wrong because the total weight of the final spanning tree depends on the partitions of V_1 and V_2 in each recursion. In each partition, we remove some edges which are in E while not in E_1, E_2. However, these edges may have smaller weights than those of edges in E_1, E_2. By removing these edges, we potentially remove the edges which could be in the minimum spanning tree. Hence, this algorithm is not correct.

Problem 5

Solution:

If the algorithm is implemented correctly, $G = (V, E)$ should have the following qualities:
1) Because in the *initialize* step, we assign the value of $s.d$ and $s.\pi$ to be 0 and NIL, we first check the value of source: $s.d = 0$ and $s.\pi = NIL$
2) In every relaxation step on edge (u, v), we should:

if $v.d > u.d + w(u.v)$ **then**
  $v.d = u.d + w(u,v)$;
  $v.\pi = u$;
end

so at the end of the algorithm, we should have $v.d = v.\pi.d + w(v.\pi, v)$
3) Because at the *initialize* step, we assign the value of all vertices $v.d = \infty$. During each iteration, if one vertex u can be reached, we would update $u.d$. So vertices which cannot be reached should have no *Predecessor*. Hence, $v.\pi = NIL$. And according to the No-path Property, if there is no path from s to v , then we always have $v.d = \delta(s,v) = \infty$.
Hence, the graph $G = (V, E)$ should meet the requirements that $v.d = \infty$ iff $v.\pi = NIL$ for all $v \neq s$.
we should check all these three qualities for each vertices. If any conditions are compromised, then we can get the conclusion that this path is incorrect. Because there are $|V|$ vertices and the runtime of each checking is $\mathcal{O}(1)$, the runtime of this part is $\mathcal{O}(V)$.
4) If all three requirements are met, we should use the checking operation in BELLMAN-

FORD algorithm to check graph $G = (V, E)$.

for *each edge* $(u, v) \in G.E$ **do**
  if $v.d > u.d + w(u, v)$ **then**
    return incorrect ;
  end
end
return correct ;

Because at the termination, according to the triangle inequality, we should have all edges $(u, v) \in G.E$,

$$\begin{aligned} v.d &= \delta(s, v) \\ &\leq \delta(s, u) + w(u, v) \\ &= u.d + w(u, v), \end{aligned}$$

Because we need to check each edge once, so the runtime of this part is $\mathcal{O}(E)$. Hence, the total runtime is $\mathcal{O}(V + E)$.

If for all the input, all the above requirement are met, then the algorithm is correct. If (1) passed, it guaranteed that the source in output is correct. If (2) passed, it guaranteed that $v.d$ is the distance from source s passing $v.\pi$ to v. If (3) passed, it guaranteed that the vertices in G which cannot be reached is well defined in output. If for all vertices, (4) passed, according to the greedy property of the problem mentioned in the class, it means that for every v, we cannot find a vertex which can lead to a smaller $v.d$. Hence, according to (2), the path which contains $v.\pi$ is the shortest path from source s to v. Because these four properties hold for every vertices and edges. $v.d$ is the shortest distance from source s to vertex v. In conclusion, the algorithm will return a correct result if all these four properties hold.