Massachusetts Institute of Technology
6.854J/18.415J: Advanced Algorithms
David Karger

Handout 4
Fri, September 14, 2012

Problem Set 1 Solutions

Problem 1. Suppose we have a Fibonacci heap that is a single chain of $k-1$ nodes. The following operations make a chain of length k. Let min be the current minimum of the Fibonacci heap:

1. Insert items $x_1 < x_2 < x_3 <$ min in an arbitrary order.
2. Delete the minimum, which is x_1 .
3. Decrease the key of x_3 to be $-\infty$, i.e. the minimum.
4. Delete the minimum, which is $x_3 = -\infty$.

The second step is the key one: it removes x_1, joins x_2 and x_3 as a chain, and then joins the original chain with the chain containing x_2 and x_3 (obtaining a tree where x_2 is the root, with x_3 and the original $(k-1)$-nodes chain as children). The third step just removes x_3 from the chain, and the last step completely deletes it. The result is that x_2 is now the root of the original chain, so we have constructed a chain of length k. For the base case, just insert a single node.

Thus, we obtain a k-node chain with $O(k)$ operations; therefore, we can construct an $\Omega(n)$-node chain with n operations.

Note that the **decrease-key** operation was essential for obtaining $\Omega(n)$ depth: without it, you can only obtain binomial heaps (which have logarithmic depth).

Problem 2. For each node, we store a counter of how many of its children were removed (call $count_i$ the counter of node i). To analyze the running time of the operations, we use the following potential function: $\Phi = \#roots + \frac{2\sum_i count_i}{k-1}$.

The **insert** operation has $O(1)$ amortized cost. Note that ϕ increases by 1 unit as in the case of the original Fibonacci heap. Thus, the cost of insert does not change.

The **decrease-key** operation will have a lower amortized cost. Suppose there are c cascading cuts. Then, the amortized cost of **decrease-key** is $1 + c + \Delta\Phi$, with $\Delta\Phi = c + \frac{-2c(k-1)+2}{k-1}$. Concluding, the cost of **decrease-key** is $1 + c + c - 2c + \frac{2}{k-1} = 1 + \frac{2}{k-1}$. Note that in the original Fibonacci heaps, this cost was $1 + 2 = 3$.

Conversely, the **delete-min** operation will have a higher amortized cost. The analysis is the same as in the case of the original Fibonacci heaps. Thus, the amortized cost of **delete-min** is bounded by the maximum degree of a heap in our data structure.

To analyze the maximum degree, we use arguments similar to those used for the original Fibonacci heaps. Let S_m be the minimum number of nodes in a heap with degree m. We will try to find a recurrence formula for S_m and then lower bound S_m. Consider a node with degree m. Then the degree of its ith child is at least $\max\{0, i-k\}$. Considering that $S_m = m+1$ for $m = 0 \ldots k-1$, we have that $S_m = k + \sum_{i=k}^{m} S_{i-k}$ for $m \geq k$. Next, note that $S_m - S_{m-1} = S_{m-k}$. The solution for this recurrence is $S_m \geq \Omega(\lambda_k^m)$, where λ_k is the largest solution to the characteristic equation $\lambda^k - \lambda^{k-1} - 1 = 0$ (note that in the case of $k = 2$, the largest solution, λ_2, is the golden ratio). If M is the highest possible degree of a heap, then we have that $S_M \leq n$, meaning that $M \leq O(\log_{\lambda_k} n) = O\big(\frac{\log_{\lambda_2} n}{\log_{\lambda_2} \lambda_k}\big)$. Thus, our modification slows down the running time of **delete-min** by a factor of $\log_{\lambda_2} \lambda_k$ (λ_k is a decreasing function of k).

Note: a common mistake was to take a potential function that gives suitable amortized cost of one operation. Remember that if you use a potential function, you have to check the running time of all operations using the same potential function.

Another common mistake was to use for the analysis the same potential function as was used for the original Fibonacci heaps. That function does not give you a lower amortized cost for **decrease-key** (consider the case when there are no cuts).

Problem 3. **(a)** We can augment the priority queue P with a linked list l. We modify the insert operation so it just puts the element in the linked list l. Now we define a consolidate operation that adds the elements of the linked list to the priority queue. We do this by creating a new priority queue P' containing only the items in the linked list l using make-heap. This takes $O(m)$ time, where m is the size of the linked list. We then merge the two queues P and P' in $O(\log n)$ time. Therefore, the total consolidation time is $O(m + \log n)$. We modify **delete-min** to first consolidate, and then call the original **delete-min**. We modify merge to first consolidate each of the augmented priority queues, and then call the original merge.

Consider a set of initially empty augmented priority queues $\{P'\}$ (that may be merged later) on which all operations are performed. The potential function Φ' is defined as the sum of the size of the lists of each of the priority queues P'. Note that inserting in a particular priority queue takes $O(1)$ amortized time. **Delete-min** on any particular priority queue also takes only $O(\log n_h)$ amortized, time, where n_h is the size of that priority queue, since the $O(mh + \log n_h)$ real work to consolidate is decreased to amortized $O(\log n_h)$ by the potential from the queue before **delete-min** was processed. Now, consider the amortized time to merge two of the augmented priority queues. We spend amortized time of $O(\log n_h) + O(\log n_{h'})$ to consolidate each one, plus the real work of merging the two priority queues which takes $O(\log n_h)$ time, assuming $n_h > n_{h'}$. The total amortized time, then, is $O(\log n_h)$.

(b) The basic idea is to use a heap of heaps (together with a list as in part (a)). The data structure is composed of several binary heaps $P_1, \ldots, P_k$ and a "master"binary heap M . The heaps $P_1, \ldots, P_k$ contain the elements of the data structure. The heap M contains as its elements the heaps $P_1, \ldots, P_k$, which are keyed (compared) according to the values of their roots.

To **insert** an element into the data structure, we just add it to the linked list l.

Delete-min first does a consolidation (which takes $O(m + logn)$ time, where m is the length of l). Then, **delete-min** retrieves the "smallest" heap P_i from M. The root of P_i is the minimum element in the data structure. Remove the minimum from P_i (usual heap operation). If P_i is not empty, insert the modified P_i back into M.

In the consolidation step, we construct a binary heap P_{k+1} from l (if l is non-empty) and empty the list l. This can be done in $O(m)$ using a standard heap construction algorithm. To finish the consolidation step, we insert P_{k+1} into into M.

To analyze the running time, let the potential function be equal to the length of the list l. Then, insert takes $O(1)$. Consolidation takes $O(m + \log n)$ real time, but $O(\log n)$ amortized time (since the length of the list decreases by m). **Delete-min** takes $O(\log n)$ time (note that the depth of all heaps, $M, P_1, \ldots, P_k$ is always $O(\log n)$).

Problem 4. This problem is open. In a previous year Eric Price (current grad student at MIT) was able to prove an $O(\log^2 n)$ time bound for delete min.

Problem 5. **(a)** Consider the offline algorithm: we process nodes in postorder (i.e., we traverse the nodes using DFS, and process a node only after processing all of its children). When we process a node a, we answer queries (a, b), such that b was processed earlier than a by doing a find in our union-find data structure D; the "name" of the result is the answer to the query. Then we union a with the parent of a, and set the name of the set-representative to be the name of the parent.

The relationship to persistent data structures is as follows. We view the order in which we process the nodes as time. Note that changes to the union-find data structure D occur exactly at the times the nodes are processed, so that we can think of the data structure as changing over time: $D_1, D_2, \ldots, D_n$. Suppose we run the above algorithm, but at each time t we process a node, we save the state of D_t. Now, suppose we wish to answer a query of the form (a, b). Suppose b was processed after a at time t. Revert to the data structure D_t , and do a "find" of a. This would answer the query (a, b).

The goal, then, is to design a persistent version of the union-find data structure

to support the following two operations:

- **find**(x, t): Find the name of x's component at time t.
- **union**(w, p, t): Union the component with name w and the component with name p at time t.

We use the disjoint-forest implementation of the union-find data structure using the union by rank heuristic. For each node, the parent pointer will also store the timestamp t at which the parent pointer became non-null (note that this occurs exactly once for each node in the tree). Furthermore, at each node we keep an extendable array, which keeps track of the updates to the "name" field, sorted by time. and the name of the parent component p involved in the union.

Therefore, to do **find**(x, t), we walk up the parent pointers in the union-find data structure until we find a node whose parent pointer became non-null at a time later than t. We don't follow any more parent pointers because they did not exist at time t. At this node, we perform a binary search on its array to locate the correct name, and this is the desired LCA. Following parent pointers takes $O(\log n)$ time due to union by rank, and the the binary search takes $O(\log n)$ time, so in total the find operation takes $O(\log n)$ time.

To do a **union**(w, p, t), we first do a **find**(w, t) and a **find**(p, t). Then we do union by rank, timestamp the edge added with t. Then, we append to the end of the array stored at the new root node, storing the parent p, along with the time. It is clear that the union operation takes $O(\log n)$ time, so the preprocessing time takes $O(n \log n)$ time.