CPSC 629: Analysis of Algorithms, Fall 2003

Solutions to Homework 3

Solution to 1 (19.2-10)

The first three operations have running time $\Theta(\log n)$ because they all depend on the size of the largest tree in the heap. Let T_0 be the largest tree in a binomial heap. We know that T_0 has at least $n/2$ nodes, the root of T_0 has degree $\Theta(\log n)$, and T_0's longest path has length $\Theta(\log n)$. If the minimal node v happen to be the root of T_0, BINOMIAL-HEAP-EXTRACT-MIN needs to insert all children of v into the linked list, taking time $\Omega(\log n)$. Similarly, it takes time $\Omega(\log n)$ to decrease the key at the end of the longest path of the T_0. Deleting the same key takes at least as much time.

For the other three operations, however, the running time is not always $\Omega(\log n)$, because they all depend on the number of trees in the heap, which can be constant even for arbitrarily large n (i.e., when n is a power of 2), but can also be $\Theta(\log n)$ for arbitrarily large n (i.e., when n is one less than a power of 2).

Comments on common errors: Giving a concrete instance does not prove an asymptotic bound. Because the asymptotic notation can conceal a constant that is large enough to cover the cost of any given instance or small enough to cancel out any large function on a given n, we cannot tell an asymptotic bound from one or a set of concrete instances. For example, someone claims that a function f is $\Omega(\lg n)$ by showing that $f > 6$ in an instance where $n = 32$, but f could be a constant 7, which is $\Theta(1)$.

□

Solution to 2 (19-1)

- a. To find the smallest key, start from the root r, descend to the child x that has $small[x] = small[r]$, and continue downward until reaching a leaf.
- b. First decrease the key of the leaf x. Suppose x's parent is y. If $k < small[y]$ then update $small[y]$, and do so in a bottom up fashion until reach the root or some node z such that $small[z] \leq k$.
- c. Start from the root r. If $k < small[r]$, then update $small[r]$, and recursively insert k into one of its children and update its *small* value; otherwise, insert k into the child x of root with largest $small[x] \leq k$ among all children of root. Do the same until the leaf level is reached, where a new node is created for k, and a pointer is add to its parent (the preceding node in the path). If its parent is full, split it, and do the bottom-up cascade splitting as in ordinary B-tree. The value *small* of new nodes are updated when the new node is created. Update the height of the root if necessary.
- d. Since the pointer to the target leaf is given, the deletion is done in a bottom-up fashion. Remove the target leaf and the pointer to it from its parent, and update its parent's *small* value. If its parent becomes empty, merge it with its sibling, and update the *small* value of the merged node. Do the bottom-up cascade merging as in the ordinary B-tree. The value *small* of merged nodes are updated when merging happens. Update the height of the root if necessary.
- e. First call MINIMUM to find out the leaf with smallest leaf, then call DELETE to remove it.
- f. To union two 2-3-4 heaps H_1, H_2, compare the height of the two heaps. Let the heights be h_1, h_2. If $h_1 > h_2$, than travel down the heap H_1 to a node n_0 with height $h_2 + 1$. Make the root of H_2 a child of n_0, doing a bottom-up cascade splitting if necessary. Again, update the value *small* for each node encountered and update height of the root if necessary. If $h_1 = h_2$, then create a new root with height $h + 1$, and value $small = \min\{small(root(h_1)), small(root(h_2))\}$, let H_1 and H_2 become children of the new root. The case $h_1 < h_2$ is symmetric to the case $h_1 > h_2$.

Comments on common errors: The 2-3-4 heap consists of a single tree, not a set of trees.

□

Solution to 3 (20.2-3)
We will prove by induction that the subtree rooted at any node of degree x has $f(x) \geq 2^x$ nodes. When $x = 0$, the subtree has 1 node. Assume $f(x) \geq 2^x$ for $x < k$. For $x = k$, the node with degree k is formed when linking two subtrees rooted at nodes of degree $k - 1$. By inductive assumption each one of these two subtrees has at least 2^{k-1} nodes. Therefore the subtree rooted at a node with degree k has at least $2^{k-1} + 2^{k-1} = 2^k$ nodes. This proves that the subtree rooted at any node of degree x has at least $f(x) \geq 2^x$ nodes. It follows that the maximum degree in a n-node Fibonacci heap is at most $\lfloor \lg n \rfloor$.

□

Solution to 4 (20.2-5)
We will prove that if there exists an implementation such that all the mergeable heap operations can run in amortized time $O(1)$ using only comparisons of keys, then we can do comparison based sorting in time $O(n)$ which is impossible as we already know. The proof is a reduction from sorting to mergeable heap operations. Given a set of n keys, insert them into a heap, and do EXTRACT-MIN repeatedly until the heap is empty. The above procedure has $O(n)$ steps, and solves the sorting problem. If all the mergeable heap operations are implemented in amortized time $O(1)$, then comparison-based sorting can be solved in time $O(n)$.

Comments on common errors: In order to show the claim is true for *arbitrary* implementation, you cannot make any assumption on what the implementation will look like. *Reduction* is one of the common techniques for this type of problems. A typical reduction have the following form:

> If there exists a solution for problem A with property C, then this solution can be used to give a solution for problem B with property D. But we know B doesn't have a solution with property D. This implies that A doesn't have a solution with property C.

□

Solution to 5 (20.2.a)
The case when the value k is less than or equal to $key[x]$ is the same as DECREASE-KEY. The amortized cost is $O(1)$.

The operation of increasing key cannot do better than DELETE, because we can reduce DELETE to increasing key. Given a target node to be removed, we can simply use the CHANGE-KEY operation to increase the key of the target node to ∞. The target node will become a node with no child. Removing such a node takes only constant time.

With this in mind, we can implement increasing key by first deleting the target node, and then reinserting it with the new key. The amortized cost is $O(\lg n) + 1 = O(\lg n)$.

Since the CHANGE-KEY operation is built using the existing operations, the amortized costs of other operations remain the same.

Comments on common errors: In order to make sure that amortized costs of other operations remain the same, you should not introduce a new operation that changes the data structure, unless you re-analyze the amortized costs of the original operations. A safe approach is to call the original operations.

□

Solution to 6 (21.4-2)
We will prove by induction that any node of rank x has at least $f(x) \geq 2^x$ descendants (including itself). When $x = 0$, the node has 1 descendant. Assume $f(x) \geq 2^x$ for $x < k$. For $x = k$, the rank k node first becomes rank k when two nodes of rank $k - 1$ are linked. By the inductive assumption, each of these two nodes has at least 2^{k-1} descendants. Therefore the node with rank k has at least $2^{k-1} + 2^{k-1} = 2^k$ descendants. This proves that any node of rank x has at least $f(x) \geq 2^x$ descendants. It follows that every node has rank at most $\lfloor \lg n \rfloor$.

Comments on common errors: The rank of a node is an upper bound of height of the node. But showing that every node has height bounded by a certain value does not imply its rank is also bound by the same value.

□

Solution to 7 (21.4-4)

We only need to show that every operation takes $O(\lg n)$ time. This is easy given the result of 21.4-2. MAKE-SET takes $O(1)$ time. Recall that the rank of a node is an upper bound on the height of the node. This implies that every tree has height at most $\lfloor \lg n \rfloor$. FIND-SET in a tree of height at most $\lfloor \lg n \rfloor$ takes $O(\lg n)$ time. UNION is a combination of two FIND-SET and one LINK, which takes $O(\lg n)$ time in total. This finishes the proof that a sequence of m operations take $O(m \lg n)$ time.

□

Solution to 8 (21-2)

- a) Consider a sequence of operations given as follows: Start from an empty data structure. Whenever an element is received, do a MAKE-TREE, then GRAFT the new tree to the node containing the next most recent element, and do a FIND-DEPTH for the new element. This procedure essentially builds a chain out of all the elements received. This sequence of m operations takes time $m+m+1+2+\ldots+\lfloor m/3 \rfloor = \Theta(m^2)$.
- b) MAKE-TREE creates a new tree with a single node and sets its pseudo-distance to 0.
- c) FIND-DEPTH is similar to FIND-SET. Starting at the target node v, it follows the path from v to the root r, summing up the pseudo-distances to compute the depth of v. After the root r is reached, it traces back along the path and does the path compression for each node x on the path: set the parent of each node on the path to be the root r and update their pseudo-distance $dist[x] = depth(x) - dist[r]$, where $dist[x]$ is the pseudo-distance of a node x, and $depth(x)$ is the actual distance of x.
- d) GRAFT(r, v) is similar to UNION by rank. It first does FIND-DEPTH on both r and v. Assume $r \in S_1$ and $v \in S_2$. Then it compares the ranks of $root(S_1)$, and $root(S_2)$, and makes the tree with smaller rank a child of the root of the larger one. (**1**) If $rank(r) < rank(v)$, then it makes r a child of $root(S_2)$, and increases r's pseudo-distance: $dist[r] = dist[r] + depth(v) + 1 - dist[root(S_2)]$. (**2**) If $rank(r) > rank(v)$, the situation is slightly more complicated. We are supposed to make r a child of v, but we actually make $root(S_2)$ a child of r. Therefore, we need to increase r's pseudo-distance: $dist[r] = dist[r] + depth(v) + 1$, and decrease $root(S_2)$'s pseudo-distance: $dist[root(S_2)] = dist[root(S_2)] - dist[r]$. Finally, (**3**) if $rank(r) = rank(v)$, then make r a child of $root(S_2)$, increase r's pseudo-distance: $dist[r] = dist[r] + depth(v) + 1 - dist[root(S_2)]$, and increase $root(S_2)$'s rank by 1.
- e) The algorithm given above is essentially the same as the disjoint set algorithm in the textbook. Following a similar analysis, the bound for worst case time complexity is $O(m \cdot \alpha(n))$, which is tight according to the notes at the end of chapter 21.

Comments on common errors: Pay attention to the different cases of GRAFT and how the pseudo-distances are updated.

□

Note: The solutions given here are terse, and in some cases, incomplete. Your answers should be complete and have more details. But you will lose points if they are too long or too complex.