P1:9 P2:8 P3:10

P4: 10

37

6.854 **Dan Ports**

Problem Set 1

2004/04/19 Collab: {sarahl,bhebert,pramook}

9

Problem 1: We will demonstrate that a heap-ordered tree of depth $\Omega(n)$ can be produced by a sequence of n Fibonacci heap operations. To do so, we introduce the procedure EXTEND, which increases the height of a degree-1 tree.

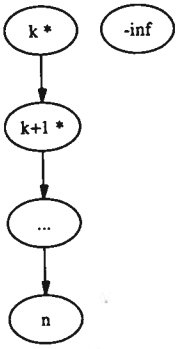
EXTEND(H)

1 ▷ precondition: H is a Fibonacci heap consisting of a degree-1 tree of height n
2 ▷ containing values greater than k, and a singleton node with value $-\infty$.
3 ▷ postcondition: H is a Fibonacci heap consisting of a degree-1 tree of height $n+1$
4 ▷ containing values greater than $k-1$, and a singleton node with value $-\infty$.
5 INSERT($H, k-1$)
6 $x \leftarrow$ INSERT(H, ∞)
7 DELETE-MIN(H)
8 INSERT($H, -\infty$)
9 DELETE-MIN(H)
10 DECREASE-KEY($H, x \rightarrow -\infty$)

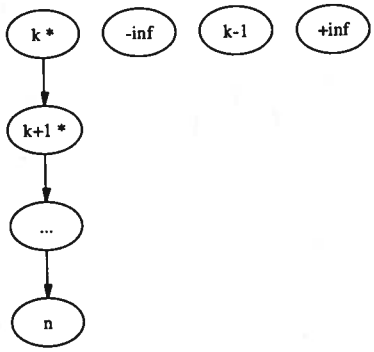
don't actually need since tree will already be consolidated as you want

We first justify the correctness of EXTEND.

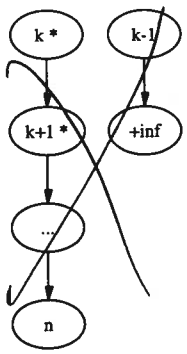
If the precondition holds, then the tree is in a state as below. Note that any of the nodes may be marked (as denoted by the *).



After the first set of insertions (lines 5 and 6), the $k-1$ and $+\infty$ nodes are added as singletons, and the heap looks like:

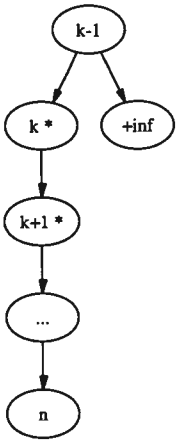


After the DELETE-MIN, the $k-1$ and $+\infty$ nodes are consolidated:

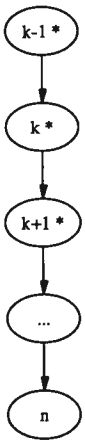


like this

Inserting another $-\infty$ and deleting the minimum consolidates the two degree-1 trees:



Finally, decreasing the $+\infty$ node to $-\infty$ removes it from the tree, marking its parent. The result is a degree-1 tree that is one level deeper:



Thus, we have shown that the procedure EXTEND meets its specifications: it increases the depth of a degree-1 tree in a Fibonacci heap by 1. Note that the number of heap operations required to perform this operation is constant.

Therefore, to create a tree of depth n, we begin by INSERTing two nodes, n and $n-1$, plus a $-\infty$ node. We then call DELETE-MIN, which removes the $-\infty$ node and consolidates the other two, creating a single degree-1 tree of depth 2. If we INSERT another $-\infty$ node, we will satisfy the precondition of EXTEND. We can then apply EXTEND $n-2$ times, resulting in a tree of depth n (plus a stray $-\infty$ node, which can be removed with a single DELETE-MIN). The correctness of this algorithm follows by induction from the correctness of EXTEND. Note that it requires a constant number of heap operations before calling EXTEND, and $\Theta(n)$ applications of EXTEND which require a constant number of heap operations each. Thus, the algorithm requires $\Theta(n)$ heap operations overall, or, equivalently, n heap operations results in a tree of depth $\Omega(n)$.

Problem 2: We modify Fibonacci heaps so that a node is cut only after it loses k children (as opposed to the standard case, where $k = 2$). To do so requires nodes to keep a counter (up to k) of the number of children that have been cut, rather than a simple mark bit. The counter is incremented each time a child is cut in a DECREASE-KEY procedure; when the counter reaches k, the parent node is cut and its counter reset to 0. We claim that this increases the amortized cost of DELETE-MIN by a constant factor, decreases the amortized cost of DECREASE-KEY by a constant factor, and does not change the amortized cost of INSERT.

To do so, we redefine the potential function. Let $\phi(i)$ be the value of the counter at node x, i.e. the number of children of x that have been cut. Then define the potential function for the data structure as follows:

$$\Phi = \#\text{roots} + \frac{2}{k}\sum_i \phi(x)$$

should be $(k-1)$ $\left[\frac{1}{k}\right.$ does not give constant decrease factor

The procedure for inserting a new node into the heap remains unchanged, since it simply involves creating a singleton tree containing the node. This is a constant amount of work. It also adds 1 to the number of roots, causing the potential to increase by 1; the $\phi(x)$ term of the potential remains unchanged since no nodes are cut. Thus, INSERT still operates in amortized constant time.

Ignoring cascading cuts, the real cost of DECREASE-KEY remains unchanged: constant time to cut the node and decrease its key. A cascading cut may be performed if a parent node x has already had k children removed from it. In this case, $\phi(x) = k$, and so the potential due to x is at least $\frac{2}{k}\phi(x) = 2$. As with normal Fibonacci heaps, this corresponds to one unit of potential to do the work of performing the cut, and one unit to counteract the increase in potential due to x becoming a new root.

-1

To analyze DELETE-MIN, we must consider the maximum degree of the heap. Note that the ith child to be added to a node x now has degree at least $i-k-1$ instead of $i-2$ since it had degree $i-1$ at the time it was joined to x, and may have lost k children since. We let S_d denote the minimum number of descendants of a node of degree d, and for convenience define $S_d = 1$ for $d \leq 0$. We then find the recurrence

$$S_d = \sum_{i=1}^{d} S_{d-k-1}$$

smaller decrease

The terms in the recurrence are of lower degree as k increases, so the root of the characteristic polynomial will be ~~larger~~. This corresponds to an increase in the base of the exponent in the solution to the linear recurrence. Since this is the minimum number of elements of a heap of degree d, the maximum degree of a heap with n elements is $\Theta(\log n)$, and the constant factor decreases as k increases (and hence the base of the exponent in the solution to the linear recurrence) increases. The time required to perform a DELETE-MIN operation is proportional to the maximum degree of the heap, so increasing k ~~improves~~ the runtime of DELETE-MIN by a constant factor.

slows down

-1

Problem 3:

Part a Suppose we are given a priority queue that can perform INSERT, DELETE-MIN, and MERGE operations in $O(\log n)$ time, and MAKE-HEAP in $O(n)$ time. We show that we can perform INSERT in $\Theta(1)$ amortized time, and DELETE-MIN and MERGE in $\Theta(\log n)$ amortized time.

We maintain a heap and an inserted list, and define the potential function Φ to be the size of the inserted list.

To INSERT a node, we add it to the inserted list. This requires constant time, and increases the potential by 1, a constant.

To perform a DELETE-MIN or MERGE, we first consolidate the inserted list into the heap. We do so by performing a MAKE-HEAP on the inserted list. This requires linear time in the size of the inserted list, but correspondingly decreases the potential by the size of the inserted list, thus incurring no amortized cost. We next MERGE the resulting heap with the original heap. This requires time logarithmically proportional to the size of the larger of the two heaps, which is bounded above by the total number of nodes. We can then apply either the DELETE-MIN or MERGE procedure to the resulting heap, requiring $O(\log n)$ amortized time. (Note that when performing a MERGE, we apply the consolidation procedure to both of the input priority queues; of course, this does not change the runtime characteristics.)

Part b We now show that we can achieve the same runtime characteristics using binary heaps, without the MERGE procedure. To do so, we use a different data structure: a binary heap of binary heaps, plus an inserted list. In the heap, the subheaps are ordered based on their minimum element. Again, we define the potential function Φ to be the size of the inserted list.

As before, to INSERT a node, we add it to the inserted list. This requires constant time, and increases the potential by 1, a constant.

Performing a DELETE-MIN is more complex. We consolidate the inserted list into the heap by performing a MAKE-HEAP on the inserted list, the inserting the resulting heap into the heap of heaps. The MAKE-HEAP requires linear time in the size of the inserted list, but incurs no amortized cost because it also decreases the potential by the size of the inserted list. The heap insertion can be performed in $O(\log n)$ time, as the heap of heaps is a binary tree whose size is bounded above by the total number of nodes in the priority queue (since there are no empty heaps in the heap of heaps).

We can then perform the actual procedure of deleting the minimum element. To do so we perform a DELETE-MIN on the heap of heaps, requiring $O(\log n)$ time, then a DELETE-MIN on the resulting heap, again requiring $O(\log n)$ time. The element extracted will be the minimum in the priority queue (we justify this below). The heap that we extracted from the heap of heaps can then be re-inserted into its new position, again requiring $O(\log n)$ time. Thus, a DELETE-MIN can be performed in $O(\log n)$ amortized time.

We now justify the correctness of the heap of heaps data structure. First, note that finding the minimum element of a binary heap is a $\Theta(1)$ operation, so there are no hidden runtime costs due to this. Next, we show that it is valid to build a heap of heaps using the minimum-element ordering. Note that the heaps are never modified without removing them from the heap of heaps, so the normal heap algorithm can order them correctly. By definition of the ordering, the minimum element of the minimum element of the heap of heaps will be the minimum element in the priority queue (ignoring the inserted list). Thus, the algorithm gives the correct result.

Problem 4: We present an algorithm for solving the online least-common-ancestor using persistent data structures. The problem can be solved online if we are able to perform a FIND on one of the nodes in the query pair at the time the other node was inserted in the postorder traversal of the tree, and find the associated *name* field. To convert nodes to "timestamps", we maintain a mapping from each node to the time it was inserted during the post-order walk; this requires $\Theta(n)$ time and space.

We augment the dynamic connectivity data structure presented in CLRS 21 (using only the union-by-rank heuristic, not path compression) to support persistence in order to achieve this goal. Note that the only changes to the data structure are a series of UNION operations which change the name of the sets they affect. In the disjoint forest implementation with union-by-rank, each UNION adds a pointer from one set representative to another, and performing a FIND follows the chain of pointers. If in addition to each pointer, we also store the time at which the UNION operation was performed, we can then perform timestamped queries by following the chain of parent pointers until a pointer is reached that has a timestamp greater than the query time. This identifies the set that the queried node was in at the specified time.

To identify the name associated with this set, we maintain a history of what names have been associated with each representative node in the node itself. We store this list as a balanced binary tree, and update it during each UNION operation, when the name changes. To find out what name was associated with the set at a particular timestamp, we search the history tree for the latest name change that happened before the requested timestamp. The size of the tree is bounded above by the number of possible timestamps, which is the number of nodes, so this requires $O(\log n)$ time.

Adding this additional information does not change the structure of the tree. Since the union-by-rank heuristic is in use, traversing the chain of parent pointers will take $O(\log n)$ time; our changes add only a small constant factor. Thus, FIND and UNION operations can be executed in $O(\log n)$ time. Building the persistent data structure is performed using the same procedure as in the offline algorithm; it requires $O(n \log n)$ preprocessing time. Queries can be performed by ordering the query pair by their ordering in the postorder traversal, then looking up the later node at the timestamp corresponding to the earlier node's insertion; if this fails, then look up the earlier node at the timestamp corresponding to the later node's insertion. This is correct because it is the procedure used by the offline algorithm; it requires $O(\log n)$ time per query.

10/10