6.854
ADVANCED ALGORITHMS
PROBLEM SET 1

JOHN WANG

Collaborators: Jason Hoch, Ryan Liu, Varun Ganesan

1. Problem 1

Problem: Unlike regular heaps, Fibonacci heaps do not achieve their good performance by keeping the depth of the heap small. Demonstrate this by exhibiting a sequence of Fibonacci heap operations on n items that produce a heap-ordered tree of depth $\Omega(n)$.

Solution: Consider the following recursive series of operations. For the ith step of the recursion, we will assume there is a tree t_1 of depth i, composed of exactly i nodes. There is also a tree t_2 which is a single node such that $root(t_1) < root(t_2)$. We shall insert two nodes a and b such that $b < a < root(t_1) < root(t_2)$. Perform a delete-min operation on the set of trees. This operation will remove b since it is the minimum of the entire structure. This leaves us with 3 roots, namely $a, root(t_1)$, and $root(t_2)$. The delete-min operation will also perform a consolidation, so that a is merged with t_1, then the resulting tree is merged with t_2.

The resulting tree has a root of a, a left child of $root(t_1)$ and a right child of $root(t_2)$. Now, we perform a decrease key on $root(t_2)$ to a value lower than a. This will cut it off from the tree, and we will be left with a tree t_1' rooted at a and t_2'. We see that t_1' will have a depth of $i+1$ and t_2' will be a single node. Thus, we are back to our original datastructure with step $i+1$ and can recurse.

Note that we performed four operations: insert a, insert b, delete-min, decrease-key. Thus, we see that after n of these operations, we will have a tree of length $n/4 = \Omega(n)$. $\square$

2. Problem 2

Problem: Suppose that Fibonacci heaps were modified so that a node was cut only after losing k children. Show that this will improve the amortized cost of decrease key (to a better constant) at the cost of a worse cost for delete-min (by a constant factor).

Solution: First we will define our potential function as $\Phi = R + 2M/(k-1)$ where R is the number of roots and M is the number of mark bits. Examining the amortized cost of insert a_i is given by:

$$a_i = c + 1 + \Delta\Phi \tag{1}$$

Where c is the number of nodes cut on a given insert (due to cascading). The real cost is $c+1$ because c nodes are cut during cascading, each requiring constant time, and $+1$ because the node must be inserted into the data structure as well. The change is potential is given by:

$$\Delta\Phi = c + \frac{2(1-(k-1)(c-1))}{k-1} \tag{2}$$

Because c nodes are cut during the cascading, an additional c roots are created which accounts for the first c term in $\Delta\Phi$. Moreover, the number of mark bits decreases by $(k-1)(c-1)$ since we cut away c nodes, which means that $c-1$ of these nodes had $k-1$ mark bits already stored which were cleared when everything was cascaded. However, we added 1 mark bit to the last node in the cascading chain, which is why we have a change of $1-(k-1)(c-1)$ mark bits. Putting this into our expression, we obtain:

$$\begin{aligned} a_i &= 1 + c + c + \frac{2(1-(k-1)(c-1))}{k-1} && (3)\\ &= 1 + 2 + \frac{2}{k-1} && (4)\\ &= 3 + \frac{2}{k-1} && (5) \end{aligned}$$

Thus, when $k=2$, the cost for insert is 5, whereas when $k>2$, the cost for insert is less than 5. Thus, the change improves the amortized cost of decrease key to a better constant.

We are left to show that cutting nodes only after losing k children makes delete-min more expensive. □

3. Problem 3

On tradeoffs in the heap operations

Problem: Let P be a priority queue that performs insert, delete-min, and merge in $O(\log n)$ time, and performs make-heap in $O(n)$ time where n is the size of the resulting priority queue. Show that P can be modified to perform insert in $O(1)$ amortized time, without affecting the cost of delete-min or merge (i.e. $O(\log n)$ amortized time). Assume that the priority queue does not support an efficient decrease-key operation.

Solution:

□

Solution:

□