Advanced Algorithms

Class Notes for Monday, October 1, 2012

Bernard Moret

Amortized Analysis (cont'd)

Fibonacci heaps (cont'd)

We have seen that the DeleteMin operation can fix up, at no extra amortized cost, the mess left behind by Insert when that operation simply creates a new root for each inserted element. In fact, what we saw can be stated more generally: the cost of cleaning up after any operation that add at most one new root (or a constant number of new roots) to the forest can be absorbed by the DeleteMin operation so as to amortize everything in constant time. Now we need to turn to the DecreaseKey operation.

We know that such an operation can be done in logarithmic time in a binary heap—it is just a matter of sifting the item up or down, as the case may be, until it has reached its proper position. However, there is no clear way of doing equivalent work in a binomial tree. Instead, we take a truly drastic measure: if the *DecreaseKey* results in making the node's key smaller than that of its parent's, thus violating the heap property, we simply *detach the node* (and its descendants) from its parent and make it the root of a new tree in the forest. As a root, the node trivially obeys the heap property. Of course, the tree from which we cut the subtree may no longer have the proper size (which should be a power of 2) nor the proper structure for a binomial queue—indeed, you can easily check that, of all the edges of the tree, only one can be cut in that manner and result in two trees (of equal sizes) that each have the proper structure. So, with this cutting approach, tree sizes are no longer limited to powers of 2, the forest is no longer equivalent to a binary representation of n, the new tree detached from its parent may be a second (third, etc.) tree of the same size in the forest, etc., etc. In fact, this cutting operation can do a lot of damage to our structure; in particular, if carried over and over with no intervening restructuring, it can remove all branching structure, leaving one child per node and transforming our trees into linked lists. Such damage is too heavy to amortize, so we must devise a way to prevent it.

We will keep a 1-bit flag with each node, indicating whether the node has previously lost a child to a *DecreaseKey* operation. (The choice of a Boolean flag rather than a counter (we could have decided to propagate the cut after the loss of some c children for some constant c) is just a matter of simplicity.) If we now remove a second child from that node—i.e., if the node's flag is set, we will cascade the process upwards: after detaching the child from the node, we will also detach the node from its own parent (which in turn, of course, could cascade higher, if the node's own parent already has its flag set). The cascading effect means that *DecreaseKey* is no longer a worst-case constant-time operation, but we will show that it amortizes to constant time per operation. The flag of a node is reset once that node is made a root—the idea here is that roots (nd only roots) can gain new

children, thus making up for the previous loss. We must be able to show that the trees used in the forest remain nice and "bushy," that is, their size remains an exponential function of their height (or the number of children of their root). Define the *rank* of a node to be the number of children (not descendants) it has; in a binomial tree, a tree with a root of rank k has exactly 2^k nodes, but in a Fibonacci heap, because of possible cuts, 2^k is only the maximum size—the tree could have fewer nodes. While two trees in a binomial queue can be merged only if they have the same size, we shall relax the condition for Fibonacci heaps: two trees can be merged only if their roots have the same rank. We shall show that the size of a tree in a Fibonacci heap is always exponential in the rank of its root—i.e., that such trees are bushy.

1. Order the children of a node by the time at which they were linked to that node and consider the ith child. At the time that child was linked to our node, our node had at least $i-1$ children—at least, because some of these older children might since then have been cut by a *DecreaseKey*. The rank of the child must have been equal to that of our node and hence it too was at least $i-1$. Since then, the child might have lost one of its own children, decreasing its rank by 1, but no more (otherwise it would have been cut from our node and made into a root). Hence the rank of the ith child is at least $i-2$. See Figure 1.

2. Knowing that the ith child has rank at least $i-2$, we can write a recurrence that describes the smallest size possible for a tree with a root of rank k. We know that, if the root has rank 0, the tree has 1 node, and that, if the root has rank 1, the smallest possible tree has just 2 nodes. Now we can write

$$\text{min-size}(k) = \underbrace{1}_{\text{root}} + \underbrace{1}_{\text{oldest child}} + \sum_{i=2}^{k} \text{min-size}(i-2)$$

This is a full-history recurrence, so we telescope it to get:

$$\begin{aligned} &\text{min-size}(k) - \text{min-size}(k-1) = \\ &= \left(2 + \sum_{i=2}^{k} \text{min-size}(i-2)\right) - \left(2 + \sum_{i=2}^{k-1} \text{min-size}(i-2)\right) \\ &= \text{min-size}(k-2) \end{aligned}$$

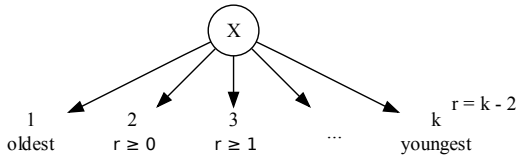

Figure 1: Relation between the size and the rank of a node.

Hence we get

$$\text{min-size}(k) = \text{min-size}(k-1) + \text{min-size}(k-2)$$

our old friend the recurrence for Fibonacci numbers. Now we know why these data structures are called Fibonacci trees—and we also know that they are bushy: their size is a Fibonacci number, which is exponential in the rank.

Thus the Boolean "cut" flag does its job: it keeps the trees bushy.

The heavy-lifting operation—the one that must fix the mess caused by a sequence of *Insert* and *DecreaseKey* operations—is *DeleteMin*. As we already noted, it has to look at every tree in the forest (by traversing the circular linked list) to identify the min element, which could take up to order n time. As we also noted, in the true spirit of amortization, we need to use this opportunity to fix the data structure and, inasmuch as possible, return it to the structure of a binomial queue—at least by reducing the number of trees to $O(\log n)$. We do this by using our size bins and forcing each bin to contain at most one tree of the appropriate size. Since, when we start, a bin may contain up to a linear number of tiny trees, we simply collect them two by two, linking each pair into a tree that now fits into the next larger bin size, continuing until we are left with nothing (in which case the bin is now empty) or with a single tree, which is the final occupant of the bin. This takes time proportional to the number of trees originally occupying the various bins (this may not be entirely obvious, since a tree can be handled multiple times, as a part of larger and larger trees: it is due to the fact that each successive bin stores trees roughly twice as large as those stored in the previous and so the total number of trees created and destroyed in this process is at most twice the original number). Thus this restructuring of the forest takes the same asymptotic time as searching for the min element—and it is done both when running *DeleteMin* and when running *FindMin* (if such an operation is supported independently). Deleting the min element is done as in the binomial queue; the number of children of any root is still $O(\log n)$, so the cost of deleting a root remains logarithmic.

Finally, *Meld* is now greatly simplified: in worst-case constant time, it simply merges the two circular linked lists (since the lists are not ordered in any way, the merge is simply altering a pair of back-and-forth pointers in each list). *Insert* remains a *Meld* of a trivial one-node tree with the current structure.

Now at last we can proceed to the amortized analysis itself. We know that the "bad" features of our Fibonacci heaps are those where they deviate from the binomial queues: extra trees in the forest and nodes that have lost children. Fibonacci heaps could have up to n trees in the forest for a heap of size n, whereas a binomial queue would have at most $\log_2 n$ trees; and of course binomial queues do not have nodes that lost children. In our potential definition, we do not consider nodes that have lost two children—for two reasons: first, they are now roots, i.e., extra trees, and so count in the potential anyway, and second, as roots, they can now acquire new children. Thus our potential takes the form

$$\Phi = \underbrace{\#\text{of trees}}_{\alpha} + \underbrace{\#\text{of non- root nodes that have lost one child}}_{\beta=2\cdot\alpha}$$

The choice $\beta = 2\alpha$ is explained below by the amortization itself. Let us then consider our two operations.

1. *DecreaseKey*: The actual cost of *DecreaseKey* is the total number of cuts, call it k; this is made of the initial cut, plus $k-1$ propagations. When propagating the cuts, every node on the propagation path is a node that lost one child and gets made into a root. Losing these $k-1$ nodes with "cut" fields set to true reduce the potential by $\beta \cdot (k-1)$, while gaining these same nodes as new roots increases the potential by $\alpha \cdot (k-1)$. We chose β as we did because we want a net decrease in potential to pay for the real cost of the operations; with $\beta = 2\alpha$, the net change in potential is $-(k-1)$ which, when added to the real cost of k, yields just 1, as desired.

2. *DeleteMin*: This operation cleans everything up. It collects all trees, places them in linked lists, one for each possible rank, then, starting at the list of lowest rank, merges the trees in the list two by two (each time taking the two trees out and placing the merged result in the list of higher rank) until at most one tree is left in the list. The result is a forest with a logarithmic number of trees, at most one at each rank. The actual cost has three pieces:

   (a) A linear search to find the root with the smallest key. If the number of trees is T, the cost for that part is (proportional to) T.

   (b) Ripping off the root takes constant time; if that root had rank r, we remove one tree from the list and replace it by r new trees. The real cost is (proportional to) r, which is in $O(\log n)$ by virtue of our Fibonacci analysis above.

   (c) Placing all trees into linked lists of trees of the same rank takes time (proportional to) $T + r - 1$; merging two by two from the list of lowest rank to the list of highest rank also takes time $T + r - 1$. (This last is less obvious, since the same tree of lower rank can figure in a logarithmic number of linkages; but note that a tree can only "lose"—that is, be linked to a new root—at most once; and when we are done, all but a logarithmic number of trees have lost.)

   The total cost is thus (proportional to) $T + r$. The effect on the potential is quite simple: the number of trees goes from T down to at most $\log n$ (the base of the logarithm is the golden ratio), so the change in potential is $-(T - \log n)$ for T larger than $\log n$, and somewhere between $-T$ and 0 otherwise. The sum of these two terms is bounded by $\log n$, which proves our claim.

Fibonacci heaps thus offer a unique operation: constant amortized DecreaseKey. However, they still have high overhead and they prove considerably slower than binary heaps for most practical applications. They do win asymptotically when the DecreaseKey operation dominates (as in Prim's algorithm for minimum spanning trees), but the asymptotic region may be beyond practical sizes.