Advanced Algorithms

Class Notes for Thursday, September 27, 2012

Bernard Moret

1 Amortized Analysis (cont'd)

1.1 Side note: regarding meldable heaps

When we saw how to meld two leftist trees, we did not discuss the other operations. But in fact, meldable heaps really have only the one operation of melding: everything else is defined in terms of it. Thus insertion is simply the melding of the current heap with a trivial heap with the single new item in it. Deletemin usually breaks the heap structure into two (or more) sub-heaps, then melds these sub-heaps back together. For leftist trees, for instance, removing the root of the tree (the item with the best priority) gives us the desired item, but leaves a left and a right subtree; both are themselves leftist trees, thanks to the recursive property, so melding them produce a single, proper leftist tree, the other produce of the Deletemin operation. We shall see other examples as we proceed.

1.2 Turning leftist trees into an amortized structure

Leftist trees are interesting, but not very useful. A balanced search tree has similar overhead, can also support Deletemin and Insert in logarithmic worst-case time, and offers more possibilities. What hinders leftist trees, when compared to, say, binary heaps, is the need to maintain extra information at every node (just as in balanced search trees). So what if we did not? That is, what if we did not have any extra information stored in a node? In that case, we could not decide whether or not to swap the children, so what could we do? The answer is deceptively simple: *swap every time!* Thus we proceed as for leftist trees, merging the two sorted lists formed by the two rightmost paths, but, when backing up, we *always* swap the two children, at every node. This has an added benefit: we can do the swapping on the way down the two lists, as we merge—there is no need to come back up, which cuts down the overhead by a significant amount. Code for this operation might look like this:

```
function Meld(p, q: PtrToNode): PtrToNode;
  (* p and q point to skew heaps. *)
  var r: PtrToNode;
  begin
    (* Call function key to avoid special cases regarding nil.
         key(nil) = infinity  *)
    (* Make things uniform--make p point to the node with smallest key. *)
    if key(p) > key(q) then swap(p,q);
```

```
 if q = nil (* At least one of the two heaps is empty. *)
   then Meld := p (* The result of the Meld is the other. *)
   else begin
          Meld := p; (* the smaller of the two roots--the smallest key *)
          r := p;
          p := p^.right;
          if key(p) > key(q) then swap(p,q);
          while q <> nil do (* p cannot be nil *)
            begin
              r^.right := r^.left; (* swap siblings on the way down *)
              r^.left := p;
              r := p;
              p := p^.right;
              if key(p) > key(q) then swap(p,q)
            end;
          r^.right := r^.left;
          r^.left := p
        end
end; (* Meld *)
```

The resulting data structure is known as a *skew heap*—like a leftist tree, it is generally skewed to the left, but there is no strict rule to be enforced at each node concerning that skew. We are now going to prove that the amortized running time of Meld is logarithmic; since Insert and Deletemin are built from a single call to Meld, they will also run in amortized logarithmic time. Skew heaps are much simpler and significantly faster than leftist trees, for the price of trading worst-case guarantees per operation (leftist trees) for worst-case guarantees for sufficiently long sequences of operations.

We shall use the potential method for the analysis. Define the weight of a node, $w(x)$, to be the number of descendants of x (including x) in the tree. A node is called *heavy* if its weight is greater than half the weight of its parent. The root of the skew heap is, by definition, not heavy. Now we define the potential of a skew heap to be (a constant times) the number of heavy nodes that are also right children. The amortized complexity result follows from the following observations:

- Only one of a sibling pair of nodes can be heavy.
  This is a trivial consequence of the definition.
- On the path from x to a descendant, y, the number of light nodes, counting y, but not x, is at most $\lfloor \lg(w(x)/w(y)) \rfloor$; in particular, any path contains at most $\lfloor \lg n \rfloor$ light nodes.
  This takes only a bit more work. If a node is light, then the subtree rooted at that node is less than half the weight of the parent tree—in other words, as we descend a path in the tree, every time we pass a light node, the weight of the tree is reduced by (at least) half. Thus, after passing through k light nodes, the ratio of weights (large to smaller) must be at least 2^k, that is, we must have $w(x)/w(y) \geq 2^k$. Solving for

k gives us the first inequality; and using $w(x) = n$ (meaning that x is the root of the entire tree) and $w(y) = 1$ (meaning that y is a leaf) gives us the second.

- During the melding operation, a heavy node can be introduced into a right path, adding to the potential of the skew heap, only if its original sibling was light. Even in this case, creation of a heavy right child is not certain, because the sibling with which it is ultimately swapped during the upward pass is necessarily heavier than the original.
  First, note that, in order to increase the potential of the skew heap, one must create a new heavy right child and new right children can arise only on the new rightmost path. Secondly, the new right child, even if it is heavy, increases the potential only if it replaces a light right child. Hence the first statement. For the second statement, it is enough to remember that the merging process does not alter the sorted order—all nodes with larger keys remain farther down the path, but may lengthen the path by adding items from the other sorted list. Thus, before the swapping on the way back up, every node on the newly formed rightmost path has at least as many descendants as it did before the merge, and possibly many more. This could turn some nodes that were light right children into heavy right children—and remember that we will then swap all children, so these will become heavy *left* children, leaving a light right child and thus not increasing the potential.
- The time spent in processing heavy nodes along the right paths can be covered by a drop in potential.
  This statement is a bit mysterious on its own, but it makes sense in the context of the second item above: since any root-to-leaf path cannot contain more than a logarithmic number of light nodes, the cost of processing light nodes is not at issue: it can only contribute $O(\log n)$ worst-case time per operation. (It does not even need to be amortized!) It is the heavy nodes on the merging paths that can prove costly—because there is no bound on their number: there could be $\Theta(n)$ of them. However, each heavy node on a rightmost path before the Meld will (i) stay heavy during the merging of the two paths and (ii) be swapped with its sibling, so that each such node will be replaced by a light node—and the constant cost to process the node is exactly compensated by a constant decrease in the potential.

We are done, except for checking initial and final potentials. In the worst case, the initial potential is 0 and the final potential is $n-2$. (Picture a tree of n items, all of them along the rightmost path—every single left pointer is nil. Then the root and the one leaf are note heavy, but all other nodes are heavy and they are all right children.) Thus we can state that a sequence of n Meld operations obeys the equation

$$\text{real time of the } n \text{ operations} \leq c_1(n \log n) + c_2(n-2)$$

and so we have proved that such a sequence of operations runs in $O(n \log n)$ worst-case time.

So, why did this work? The proof above shows that it works, but it is a bit short on intuition. Since we had no basis for deciding whether to swap the children, why did we

decide to swap every time? Couldn't we equally well chosen never to swap? What makes everything work is the second of the four observations made above: on the rightmost path (as on any path), the number of light nodes is at most logarithmic. What this implies is that long rightmost paths, which we want to avoid, are made mostly of heavy nodes and the best way to deal with heavy right children is to swap them with their sibling. In cases where such heavy nodes are few, swapping everything does not matter much: it costs us little, as the paths are short, and it does not damage the tree too much even if it brings new heavy nodes on the rightmost path; but in cases where such heavy nodes are numerous, it is crucial to swap them out of the way, or we will have to traverse long paths on Meld after Meld after Meld. Thus we might as well swap every time: there is much to gain and nothing of importance to lose in doing so.

The data structure is much simpler, but the analysis is a bit more involved than for the leftist trees. Thus there is more than one tradeoff here: we traded simplicity of design for complexity of analysis!

1.3 Improving DecreaseKey

The DecreaseKey operation (improve the priority of an element) is crucial to a number of algorithms, starting with both of the standard algorithms for constructing a minimum spanning tree, Kruskal's and Prim's algorithms. In a binary heap, it takes $O(\log n)$ time; in many pointer-based priority queues, it is not easily implementable—typically the only solution is to delete the item from the priority queue (if that is supported) and reinsert it with the new priority. Yet we can design a meldable priority queue that supports DecreaseKey in amortized *constant* time. As for leftist trees and skew heaps, the new structure is a simplified version of a worst-case design. So let us start with the original, worst-case per operation, data structure, known as a *binomial queue*.

1.4 Binomial queues

The main idea behind binomial queues is to decompose a priority queue into a forest of trees in the same way that a positive integer is decomposed into powers of two, so that melding becomes similar to a binary addition. To accomplish this, we need a tree structure that can, like powers of two, be "doubled" to produce the next larger tree. The solution was to use non-binary trees; that way, two identical tree structures can be combined in constant time simply by making one of the tree a child of the root of the other. As for binary representation, where a digit can represent only powers of two, our trees will have sizes that must be powers of two. The trivial tree ($k = 0$) is a single node; merging two such trees, by making one a child of the other, yield a tree of size 2 ($k = 1$) made of a root and one child; merging two of these yields a tree of size 4 ($k = 2$), made of a root and two children, one a tree of size 1 and the other a tree of size 2; and so forth. In general, a tree of size 2^k has a root with k children, where the ith child is a tree of size 2^i, for $i = 0, \ldots, k-1$. Each of these trees will be heap-ordered, that is, the parent of a node cannot have a worse priority than that of the node; in consequence, when merging two trees into one of twice

the size, the choice of which tree is made a subtree of the other is dictated by the keys of the two roots.

Now, a binomial queue is an array of pointers, which can be nil or point to trees of the type just described; these pointers are just like binary digits, with a nil pointer corresponding to a 0 and a pointer to a tree corresponding to a 1. Naturally, the last pointer, if not nil, points to a tree of size 1; and, in general, the *i*th pointer from the end of the array, if not nil, points to a tree of size 2^i. Thus a binomial queue of size *n* is represented by an array that corresponds exactly to the unsigned binary representation of *n*, along with the forest of trees corresponding to the nonzero digits in that representation. Note that a forest never has more than one tree of size 2^i, for any *i*.

The Meld operation is just a binary addition. Given two binomial queues, we simply start at the end of each array and look at the corresponding pointers (digits); if either one is nil, we copy the other pointer in the array for the melded queue, but if neither one is nil (both digits are "1"), we write a nil pointer in that position in the array for the melded queue, merge the two trees into one twice the size, and report it as a "carry" in the addition. In general, when working on the *i*th position from the end, we have three pointers to consider: the two pointers stored in the *i*th position from the end in each array, and a carry pointer; if at least two of these three pointers are nil, we copy the third pointer in the *i*th position from the end in the array for the melded queue and report a nil carry; if exactly one of these pointers is nil, we merge the trees from the other two pointers into a tree of twice the size and report its pointer as carry, and we write a nil pointer in the *i*th position from the end in the array for the melded queue; and if none of the three pointers is nil, we pick one (arbitrarily) to write in the *i*th position from the end in the array for the melded queue and we merge the tree from the other two pointers into a tree of twice the size and report its pointer as a carry. The work done is constant per array position and so is logarithmic in the size of the priority queue.

Inserting in a binomial queue is just melding a queue of size 1 with the existing queue, which takes worst-case logarithmic time. Deletemin requires first locating the tree with the root with smallest key, which takes a linear search through the array, checking the priority of the root of each tree in the forest, taking logarithmic time; then we remove this tree from the forest, remove and return its root (the element with smallest key), and place each of its subtrees in a new array, once again taking logarithmic time; and finally, we meld the two arrays to produce the new priority queue with one less element, yet again taking logarithmic time. So Deletemin runs in worst-case logarithmic time—although with a large coefficient due to these three phases.

Binomial queues, like leftist trees, are interesting structures, but not really competitive when it comes to Insert and Deletemin operations. So, once again, we shall remove a constraint from the structure, losing the worst-case bound per operation, but retaining an amortized bound and streamlining the structure. Since binomial queues obey several constraints (heap order on each tree, at most one tree of each size, each tree of size equal to a power of 2), there are a number of ways to relax them. The one we shall discuss is a relaxation on the last two constraints: we shall tolerate more than one tree of a given size in the forest and also trees whose size is not a power of 2. The result is known as a *Fibonacci*

heap.

1.5 Fibonacci heaps

The first modification we are going to make is to allow more than one tree of the same size. This allows us to insert a new node in constant worst-case time, simply by adding another tree of size 1 to the forest. We need a different representation of the forest, however, so we will place all roots in a circular linked list, and do the same for all children of a node. We will force the external pointer to the data structure to point to the root with smallest key, but we will not require the same for child pointers, which are allowed to point to any one of the children of the node. Because of our requirement for the external pointer, a Meld operation now takes constant worst-case time: join the two circular linked lists into one and, of the two external pointers, retain the one pointing to the node with the smaller key. This reduction in the cost of Insert and Meld, however, is at the direct expense of the Deletemin operation, since it must search through all roots (and all children of the removed node) to identify the new min element. As this can be very expensive (linear time in case we simply did a bunch of Insert operations), we will take advantage of the expense to put the data structure back into decent shape—this will increase the running time by a constant factor, but not alter the asymptotic running time. Because the cost of DeleteMin is driven for the most part by the number of trees at the root level, our goal in this restructuring will be to reduce the number of trees down to $O(\log n)$. In effect, we will simply carry out delayed operations: by not merging trees of the same size during insertion and melding, we potentially created many trees of the same size, so now, as part of DeleteMin, we carry out these mergings. We simply set up an array of logarithmic size (as in binomial queues) and start traversing the linked list of roots; at very new root, we add its tree to the array in the proper position; if that position was empty, we just place the tree in it, while, if that position was occupied, we grab the pointer, set that position to empty, merge our tree with that indicated by the pointer, and attempt to place the resulting tree (of twice the size) into the next position, repeating the process as necessary. The entire process takes time proportional to the number of entries in the circular linked list of tree roots at the beginning of the operation. That this amortizes properly is now clear: in order to have to deal with k singleton nodes in the circular linked list of roots, we must have previously carried out at least k insertions, each of which took constant time, for a total of $\Theta(k)$ time; and now we have to find the minimum element, remove it, and restore a binomial queue structure to the collection of trees, which also takes us $\Theta(k)$ operations, so the total time is $\Theta(k)$ for $k+1$ operations.

This first modification is best viewed as a *lazy insertion* strategy. (Lazy operations are those that delay corrective operations until some more opportune time.) It does nothing to speed up the DecreaseKey operation and does not fundamentally improve the data structure. The next step is to consider what can be done to implement a DecreaseKey and how to amortize its cost.