Advanced Algorithms

Lecture notes

October 7, 2009

1 Skew heaps: amortized analysis

To run an amortized analysis, we need a potential function. A potential function measures how bad the data structure is. A first guess could be to take the number of nodes in the rightmost path, but it is not possible to keep track of them since we swap them all the time. Instead, we will use a definition in the same spirit, but rougher: the number of right children that are heavy. We define a node to be heavy if the size of the subtree of which it is the root (counting the node itself) is greater than half the size of the subtree rooted at its parent—see Figure 1 for an illustration. (In the figure, the number written in each node is the size of the subtree rooted at that node.) Note that, if the tree is leftist, then the potential is zero.

In a skew heap, as in a leftist tree, everything is done using the Meld operation, thus we will show that the amortized running time of this operation is logarithmic:

\[
\text{actual time}(\text{Meld}) + \Delta \Phi \text{ is } O(\log n)
\]

Observe the following:

- At most one of a pair of sibling nodes can be heavy (since the number of descendants of both siblings cannot both be greater than half the number of descendants of their parent).

- On a path from \(X\) to one of its descendants \(Y\), we need to bound the number of light nodes (as opposed to heavy), \(k\), as a function of \(w(X)\) and \(w(Y)\) (where the weight \(w(X)\) equals the number of descendants of \(X\), including \(X\) itself).

![Figure 1: A few examples of trees with and without heavy (H) nodes.](image-url)
Figure 2: On a path from \( X \) to \( Y \) with \( k \) light nodes.

Now, if a node is light, its weight is less than half that of its parent; equivalently, the weight of its parent is at least twice that of the light child, \( w(P) \geq 2 \cdot w(LC) \). So, now, on a path from node \( X \) to some descendant \( Y \) with \( k \) light nodes along the path (a situation illustrated in Figure 2), the weight must at least double every time we move from a light child to its parent and so we have:

\[
\begin{align*}
\quad w(X) & \geq 2^k \cdot w(Y) \\
\quad k & \leq \log \frac{w(X)}{w(Y)}
\end{align*}
\]

In particular, the number of light nodes on any path cannot exceed \( \log \frac{w(\text{root})}{w(\text{leaf})} = \frac{\log n}{2} = \log n \).

So the actual time for a Meld on a rightmost path is made of two parts:

\[
\begin{align*}
\text{cost of light nodes} & \Rightarrow \text{proved } O(\log n) \\
+ \text{cost of heavy nodes} & \quad \text{(proven)}
\end{align*}
\]

We have a bound for the number of light nodes, but no bound for the number of heavy nodes on a path other—other, that is, than the trivial bound \( O(n) \).

By swapping (flipping) all the children of the nodes along the rightmost path during the Meld, we will make some heavy nodes into left children, so that they will cease to contribute to the potential. At the same time, however, the merging itself will change the left-right sibling association every time we move from one path to the other, and so could create new heavy nodes that are right children—see Figure 3 for an illustration of these new sibling pairings. In other words, any heavy node on the rightmost path of the two heaps pays for itself, since it will become a left child and cease to
contribute to the potential: constant processing cost plus constant decrease in the potential. But we must bound the number of newly created heavy right children, because these will increase the potential. Fortunately, that is not too hard: any such node is, at merging, the heavy left sibling of a now light node on the rightmost path—then the flipping occurs and the heavy nodes are placed onto the rightmost path. We have seen how to bound the number of light nodes—on any path from the root to a leaf, including the rightmost path, there are at most \( \log n \) of them. Hence the number of newly created heavy right nodes is \( O(\log n) \).

Overall, then, the amortized time of a meld is:

\[
\text{amortized cost} = \text{actual cost} + \Delta \Phi = \log n + V_{\text{light}} + (V_{\text{heavy}} + \log n)
\]

where \( V \) is the number of heavy nodes on the rightmost path. Melding two skew heaps takes therefore \( O(\log n) \) amortized time, as desired.

In a meldable heap, insert is a Meld of an existing heap with a trivial heap of only one element and DeleteMin amounts to stripping the root with the min value and melding its subtrees (at least in most cases). Even if we knew that Meld takes amortized logarithmic time, we could not immediately say that Insert or DeleteMin is also amortized logarithmic, because we do not know what happens to the potential used in the amortized analysis. We need to verify that change from initial to final potential is not dominating the actual costs. We had to do that for the analysis of Meld, of course, but it has to be done separately for the other operations, since each includes a step that modifies the structure independently of the Meld itself. In the case of skew heaps, it is easy to see that the additional work (in addition to the Meld) does not appreciably alter the potential.
2 Fibonacci heaps

Let us briefly revisit the specific steps taken to relax binomial queues into Fibonacci heaps. We decided to maintain a “cut” Boolean flag at each node, indicating whether the node in question has lost a child since the last time it was a root. The rationale here is to prevent trees from getting too “skinny”—too tall for their size, with concomitant increase in the running time. We can have any number of consecutive DecreaseKey operations, with no intervening DeleteMin to restructure the forest; a long sequence of DecreaseKey operations, if not controlled with the “cut” flag, could “trim” the trees down to a few very long branches. 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 this is indeed the intent of the “cut” flag, then we must be able to show that the trees used in the forest remain nice and “bushy.” This is the first step in our analysis. 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 \( 2^k \) nodes, but in a Fibonacci heap, because of possible cuts, \( 2^k \) is only the maximum size—the tree could have fewer nodes. We can merge two trees (as in binomial queues) whenever they have the same rank—even though they may not have the same size.

1. Order the children of a node by the time at which they were linked to that node and consider the \( i \)th 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 \( i \)th child is at least \( i - 2 \). See Figure 4.

2. Knowing that the \( i \)th 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...

![Figure 4: Relation between the size and the rank of a node.](image-url)
just 2 nodes. Now we can write

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

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

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

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

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

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.

Now we can proceed to the amortized analysis itself. (The preceding analysis is a standard worst-case analysis for the smallest size of a tree of a certain rank.) 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 $\log n$ 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 root, they can now acquire new children. Thus our potential takes the form

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

The choice $\beta = 2\alpha$ is explained 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 $\beta$ 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 $\min(0, -(T - \log n))$. The sum of these two terms is bounded by $\log n$, which proves our claim.

3 **Splay trees**

Splay trees are essentially amortized binary search trees. The basic idea is that the last item queried is likely to be queried again soon, so that we are going to use the cost of traversing down the tree to find the item so bring that item up to the root, where access will be very fast that next time. Even if the item is not in the tree, we will bring up an item from that region of the tree (the predecessor or successor of the nil pointer that stopped the search).

No other structuring effort is made: no weight- or height-balance, no notion of alternating colors, or anything else. The only property maintained is the binary search property common to all binary search trees:

$$\text{keys(left subtree)} \leq \text{key(parent)} \leq \text{keys(right subtree)}$$

The basic operation is *Splay*—search for the item and bring it (or, if absent, its predecessor or successor) to the root. Splaying is done according to specific patterns—very simple ones, unlike the rather complex patterns used
in, e.g., red-black trees. As with meldable heaps, where all operations were defined in terms of Meld, in a splay tree all operations are defined in terms of Splay. We need three patterns to carry out Splay\((x)\), repeated as needed until \(x\) is the root of the whole tree:

- **Root pattern:** if \(x\) is child of the root, the tree is rotated on the edge between \(x\) and the root, terminating the Splay. See Figure 5.

- **Zig-zig:** if \(x\) is at least two levels below the root and \(x\), its parent \(y\), and its grandparent \(z\) form a sorted list, we rotate the list completely around to bring \(x\) to the root of the pattern. See Figure 6.

- **Zig-zag:** if \(x\) is at least two levels below the root and \(x\), its parent \(y\), and its grandparent \(z\) do not form a sorted list, we bring \(x\) to the root of the pattern and make \(y\) and \(z\) its left and right children. See Figure 7.

We claim that a Splay operation on a tree of \(n\) nodes takes \(O(\log n)\) amortized time. It should be clear that a splay tree could degenerate into a linked list, in which a single operation could easily take linear time, so the worst-case running time per operation is linear. The deeper we go into the tree, however, the more patterns will be applied on the way up and the more restructuring will take place—the basic principle of any self-adjusting data structure.

Before we do the actual analysis, let us quickly check how we would run an insertion or deletion. Figure 8 shows an insertion: we splay the tree on the key to be inserted, thereby bringing up its predecessor (in this case):
once that is done, we simply attach that predecessor, with its left subtree, as the left child of the new item, and detach the right child of the predecessor to make it the right child of the new root. Thus an insertion is one Splay operation and $O(1)$ pointer manipulations.

Figure 9 shows a deletion: we splay the tree on the key of the item to be deleted, thereby bring that node up to the root; we remove that root, leaving two subtrees. We can then splay the left subtree (A in the figure) on the same key again, bringing up to the root of that subtree the predecessor of $x$ and thus leaving its right child empty; we then attach the other tree (B in the figure) as the new right child. Thus a deletion is two Splay and $O(1)$ pointer manipulations.