Advanced Algorithms

Class Notes for Monday, October 1, 2012

Bernard Moret

Amortized Analysis (cont'd)

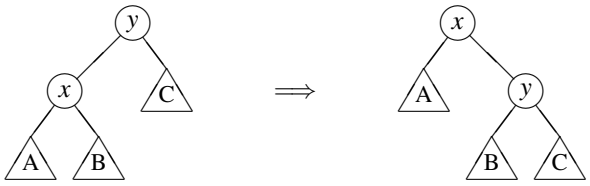
Splay trees

Our next amortized design is for a full-fledged binary search tree structure. It is inspired by the move-to-front heuristic (to be discussed next week), translated here into a move-to-root operation. We are going to maintain a binary search tree, that is, a tree where each node stores a search key, a node has a most 2 children, and any key in the left subtree of a node is no larger than the key of node, which in turn is no larger than any key in the right subtree of the node. We will maintain this tree without the help of any flag or counter, so it will not support operations that run in logarithmic worst-case time, but it will support such operations in logarithmic amortized time. Like our amortized priority queues, where there was a single defining operation (melding), our structure will have a single defining operation, named *Splay*. This operation takes a tree and a search key, searches for the key in the tree and does one of two things: (i) if there is at least one node in the tree with such a key, it takes the first such node it encounters in its search and moves it up to the root; (ii) otherwise, it takes the last node encountered in its search (which contains either the inorder predecessor or the inorder successor of the search key value) and moves it up to the root. The move-to-root is the source of amortization, but also makes the structure self-adjusting to streams of searches.

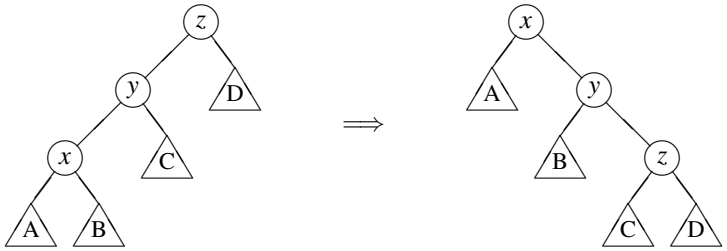
The move-to-root is done through successive upward rotations, of two levels at a time (unless, of course, the node to be moved is currently a child of the root, in which case the rotation moves the node just one level up), as illustrated in Figure 1. There is nothing complicated in these rotations: they simply do what they have to do in order to move a given node two levels (or one level) up while respecting the binary search ordering.

Splay is now the main operation, in terms of which all others are implemented. A search is just a call to *Splay* followed by a check of the key of the root. A deletion does the same, then removes the root, leaving two subtrees; it then calls *Splay* again on one of the two subtrees, which transforms that subtree into one where the root has a nil pointer, where we attach the other subtree. An insertion also starts by calling *Splay* with the key of the item to be inserted; assuming the item is not already in the tree, the operation will result in a tree where the root is either the inorder predecessor or inorder successor of the item to be inserted; we set the item in a new root and attach to it the two pieces of the tree. (These pieces depend on whether the root of the original tree is the inorder predecessor or the inorder successor of the new item; if the former, then we attach on the left the inorder predecessor and its left subtree and we attach on the right the right subtree of the inorder predecessor. Otherwise, we attach on the left the left subtree of the inorder successor, etc.)

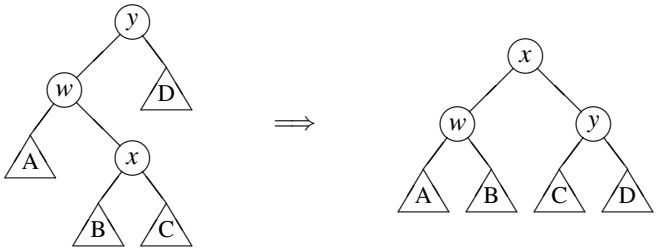
We claim that the amortized running time of the *Splay* operation is logarithmic; it will then follow (after checking initial and final potentials, just in case) that insertion, deletion,



(a) The node being accessed, x, has no grandparent.



(b) The zig-zig pattern



(c) The zig-zag pattern

Figure 1: The Three Basic Rotations Used to Implement *Splay*

and search also all run in logarithmic amortized time. Moreover, we can also use this structure (as we can any search tree) as a priority queue: *DeleteMin* simply needs to return the leftmost node, which it can do by calling *Splay* with the minimum possible key (no need to know the actual minimum in the tree), and then deleting it as described above, which again takes logarithmic amortized time. To analyze the splay operation, we need to define a suitable potential. As for skew heaps, define the weight of a node, $w(x)$, to be the total number of nodes in the subtree rooted at x and define the *rank* of x to be $\log w(x)$. We now define the potential of a tree to be the sum of the ranks of its nodes, $\Phi(T) = \sum_{x \in T} r(x)$. The rationale behind this definition is that a larger potential indicates a taller (and hence worse) tree. The root may have the largest rank of all tree nodes, but leaves may contribute more to the potential, as they contribute to the rank of each of their ancestors; and thus also the leaves farther from the root contribute more than those closer to it. We also need to settle on a measure of the actual cost of a *Splay*. To keep things simple, we just use the distance from the node to be splayed to the root of the tree. We are now ready to proceed with the proof; this proof is just algebra, but is a bit involved.

We are going to prove that, under the assumptions made above, the amortized cost of *Splay(x)* cannot exceed $1 + 3 \cdot (\lg n - r(x))$, where we chose these exact coefficients to enable us to make the proof go through. To do this, we need to account precisely for the effect on the potential of each rotation, then sum the real costs and potential changes associated with each rotation, from x up to the root.

If node x is already at the root, no rotation is performed, the potential of the splay tree remains unchanged, $r(x)$ equals $\lg n$, and our expression for the amortized bound reduces to 1. Since the depth of x is zero, our time measure for the splaying is zero, so the inequality is obeyed.

In what follows, primes will be used to distinguish ranks and weights after completion of a rotation from ranks and weights before completion of the rotation. We shall show that, when a rotation is applied, $3\big(r'(x) - r(x)\big)$ (or $3\big(r'(x) - r(x)\big) + 1$ in the case of the first rotation) is greater than the change in potential (new minus old) plus the time charged for performing the rotation. It then follows that $3\big(\lg n - r(x)\big) + 1$ is the correct amortization value to associate with a *Splay* action: the sequence of rotations results in a sequence of ranks for node x, $r(x)$, $r'(x)$, $r''(x)$, $\ldots$, $r^{(k)}(x)$, and

$$\begin{aligned} 3\big(r^{(k)}(x) - r^{(k-1)}(x)\big) + \cdots + 3\big(r''(x) - r'(x)\big) + 3\big(r'(x) - r(x)\big) + 1 & \\ = 3\big(r^{(k)}(x) - r(x)\big) + 1 & \\ = 3\big(\lg n - r(x)\big) + 1 & \end{aligned}$$

bounds the accumulated change in potential, $\Phi_i - \Phi_{i-1}$, plus the total cost of the rotations. The addition of one is only necessary if the first template is applied. Also, note that $3\big(r^{(j)}(x) - r^{(j-1)}(x)\big)$ is strictly greater than the change in potential plus the time charged for performing a basic step.

There are three cases, each corresponding to one of the three rotation templates:

1. In the first rotation, we see that only nodes x and y can change rank, so the change in potential (plus one charged to the rotation) is given by

$$1 + r'(x) + r'(y) - r(x) - r(y)$$

$$\begin{aligned} &< 1 + r'(x) - r(x) && \text{since } w(y) > w'(y) \\ &< 1 + 3\big(r'(x) - r(x)\big) && \text{since } w'(x) > w(x). \end{aligned}$$

2. In the second rotation, we see that the charge for the running time is two and that the change in potential is due entirely to changes in the ranks of nodes x, y, and z; thus we have

$$\begin{aligned} &2 + r'(x) + r'(y) + r'(z) - r(x) - r(y) - r(z) \\ &\quad = 2 + r'(y) + r'(z) - r(x) - r(y) && \text{since } w'(x) = w(z) \\ &\quad < 2 + r'(x) + r'(z) - 2r(x) && \text{since } w'(x) > w'(y) \text{ and } w(y) > w(x) \\ &\quad < 3\big(r'(x) - r(x)\big). \end{aligned}$$

   Proving the correctness of this last inequality requires a number of algebraically straightforward, but obscure, steps. The most important realization is that for s and t in the triangle bounded by $s > 0$, $t > 0$, and $s + t \leq 1$, the sum $\lg s + \lg t$ is maximized for $s = t = \frac{1}{2}$, with $\lg s + \lg t = -2$. (Clearly the maximum occurs along the line $s + t = 1$, and then the application of some simple calculus gives us this result.) After eliminating one multiple of $r'(x) - r(x)$ from both sides of the inequality, we rearrange $2 + r'(z) - r(x) < 2r'(x) - 2r(x)$ into the equivalent inequality, $\big(r(x) - r'(x)\big) + \big(r'(z) - r'(x)\big) < -2$. Now $r(x) - r'(x) = \lg\big(w(x)/w'(x)\big)$ and $r'(z) - r'(x) = \lg\big(w'(z)/w'(x)\big)$ and we see that $w(x) + w'(z) < w'(x)$ (in fact, the left-hand side is exactly one less than the right, due to the presence of the node with key y), so that we can conclude that $w(x)/w'(x) + w'(z)/w'(x) < 1$ and thus $\lg\big(w(x)/w'(x)\big) + \lg\big(w'(z)/w'(x)\big) < -2$, so that the inequality is established.

3. Proceeding as in the previous case, we get

$$\begin{aligned} &2 + r'(x) + r'(w) + r'(y) - r(x) - r(w) - r(y) \\ &\quad < 2 + r'(w) + r'(y) - 2r(x) && \text{since } w'(x) = w(y) \text{ and } w(x) < w(w) \\ &\quad < 2\big(r'(x) - r(x)\big) \\ &\quad < 3\big(r'(x) - r(x)\big), \end{aligned}$$

   where the result on the maximum of $\lg s + \lg t$ on the triangle bounded by $s > 0$, $t > 0$, and $s + t \leq 1$ is again used to establish the inequality $2 + r'(w) + r'(y) - 2r(x) < 2\big(r'(x) - r(x)\big)$.

This is the desired proof. Now we can verify that Insert can be amortized to $4\lg n + 2$ and Delete to $6\lg n + 3$, so that we can conclude that every operation on a *Splay* tree runs in amortized logarithmic time.

Because a *Splay* tree is a self-adjusting structure, it ends up being the best implementation of a meldable pirority queue. The min element is always the leftmost node in the tree; *DeleteMin* simply splays the smallest possible key value and moves this leftmost node to the root in amortized logarithmic time. After a mix of *DeleteMin* and *Insert* operations, the tree assumes a right-leaning shape, making the leftmost node closer to the root and thus improving the efficiency of the data structure.