COS 423: Theory of Algorithms Spring 2007

Lecture 4 — February 14, 2007

Prof. Robert Tarjan *Scribe: Wolfgang Mulzer*

1 Overview

In this lecture, we are going to finish the description of splay trees and start with priority queues.

2 Amortized Analysis of Splay Tree Operations

Recall that in order to acces a node x in a splay tree, we use the standard binary search tree algorithm to find x and then use the splay operation to move x to the root. We are going to show that the amortized cost of such a splay operation is $O(\log n)$.

For the analysis, we will use a potential function Φ. To define Φ, assign each node x in the tree a positive weight $w(x)$. Then we define the size of x, $s(x)$, as the total weight of x and all its descendants in the splay tree, and the rank of x, $r(x)$, as $\log s(x)$. In the simplest case, we can set all the weights to 1, then the size of x corresponds to the size of the subtree rooted in x (between 1 and n). The potential of the splay tree is defined as the sum of the ranks of all the nodes in the tree.

Recall that the amortized cost of an operation is defined as the actual cost of the operation plus the change in potential. In order to analyze a splay operation, we look at the individual splay steps and see how the potential changes. By a clever case analysis, part of which is shown in the MIT notes, we can see that the amortized cost of a zig-zig or zig-zag operation on x is at most $3(r'(x) - r(x))$ and the amortized cost of a zig operation is at most $3(r'(x) - r(x)) + 1$, where $r(x)$, $r'(x)$ denote the rank of x before and after the splay step.

For completeness, we give the details for all cases. For figures of the various cases, see the MIT notes, where a zig-zig step is called an rr or ll step, a zig-zag step is called an rl or lr step, and a zig step is called an r or l step. Let x be the node at which the step occurs, let y be its parent, and let z be its grandparent (both before the step). Unprimed values are before the step, primed values are after the step. In the case of a zig-zig or zig-zag step, the amortized cost of the step is 1 (for the actual cost of the two rotations) plus $r'(x) + r'(y) + r'(z)$ minus $r(x) + r(y) + r(z)$, since only x, y, and z can change in total weight as a result of the step. We have $r(x) < r(y) < r(z) = r'(x)$. Also $r'(x) > r'(y)$ and $r'(x) > r'(z)$. The amortized cost of the step is thus

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

In the case of a zig-zig step, we see (referring to Figure 3.1 of the MIT notes) that $s(x)+s'(z) < s'(x)$, since the subtree rooted at x before the step and the subtree rooted at z after the step are node-disjoint and together contain all descendants of x after the step except y. Thus either $s'(x) > 2s(x)$, or $s'(x) > 2s'(z)$. Taking logs and rearranging, either $r'(x) - r(x) > 1$, or $r'(x) - r'(z) > 1$. In the former case, the amortized cost of the step is less than

$$r'(x) - r(x) + r'(x) + r'(x) - r(x) - r(x) = 3(r'(x) - r(x)).$$

In the latter case, the amortized cost of the step is less than

$$r'(x) - r'(z) + r'(x) + r'(z) - r(x) - r(x) = 2(r'(x) - r(x)) < 3(r'(x) - r(x)).$$

In the case of a zig-zag step, we see (referring to Figure 3.2 of the MIT notes) that $s'(y) + s'(z) < s'(x)$, since the subtrees rooted at y and z after the step are node-disjoint and contain all descendants of x after the step except x. Taking logs and rearranging, either $r'(x) - r'(y) > 1$, or $r'(x) - r'(z) > 1$. In the former case, the amortized cost of the step is less than

$$r'(x) - r'(y) + r'(y) + r'(x) - r(x) - r(x) = 2(r'(x) - r(x)) < 3(r'(x) - r(x)).$$

In the latter case, the amortized cost of the step is less than $3(r'(x) - r(x))$ by the same argument as in the latter case of a zig-zig step.

Finally, in the case of a zig step, the amortized cost is 1 (for the rotation) plus $r'(x) + r'(y)$ minus $r(x) + r(y)$. This is at most

$$1 + 2r'(x) - 2r(x) < 1 + 3(r'(x) - r(x)).$$

Now, assume that we splay a node x and let $r_i(x)$ be the rank of x after the ith splay step $(0 \leq i \leq k)$. Since in each splay operation there can be at most one zig-step, the total amortized cost of the splay operation is at most

$$1 + \sum_{i=1}^{k} 3(r_i(x) - r_{i-1}(x)) = 1 + 3(r_k(x) - r_0(x)).$$

Given our definition of rank, this is

$$3 \log \frac{S}{s_0(x)} + 1,$$

where S denotes the total weight of all the nodes in the tree and $s_0(x)$ is the initial size of x. If all the weights are chosen to be 1, this is $O(\log n)$.

Now, what does this analysis mean? Assume that we have a tree of n nodes, and we perform m accesses with splaying. Then the total cost of this sequence of operations is the amortized cost plus $\Phi_0 - \Phi_m$, where Φ_0, Φ_m denote the initial and final potential. Since the final potential is nonnegative and the initial potential is at most $n \log n$ (using unit weights), we get a total cost of $O((m + n) \log n)$.

3 Insertion and Deletion

In order to insert a node into a splay tree, we perform standard binary search tree insertion and then splay on the inserted node. The above analysis shows that the amortized cost for the splay step is $O(\log n)$, but we must also account for the increase in potential when we insert the node. In the MIT lecture notes, it is shown that this increase is also $O(\log n)$.

Similarly, for deletion, we perform standard binary search tree deletion and splay on the parent of the deleted node. The amortized cost for the splay is $O(\log n)$, and the potential can only decrease when we remove a node.

Hence, if we start with an empty tree and perform an arbitrary sequence of m insert, access, and delete operations, the total time is $O(m \log n)$.

4 Splay Trees vs Red Black Trees, Join and Split

What do splay trees buy us compared to red black trees? Firstly, we can save the space needed for the color bits, and the restructuring rules are simpler. On the other hand, we need to restructure the tree for every access.

On splay trees, we can implement joins and splits very easily. Indeed, assume we are given two splay trees T_1, T_2 such that all the nodes in T_1 are smaller than all the nodes in T_2. Then we can join them by finding the maximum element in T_1, splaying it to the root, and setting T_2 as its right child. Splitting can be done similarly: Find the node to split on, splay it to the root and then split the tree. The amortized cost for join and split is $O(\log n)$.

5 Static Optimality and the Dynamic Optimality Conjecture

Another nice property of splay trees is that they are self-adjusting. One way of making this precise is the following: Assume we are given a sequence of m accesses, and for each element, we know how often it is accessed during this sequence. If we set the weight of node x to this frequency, then we get that the amortized cost to access x is $O(\log \frac{m}{\text{\# accesses to } x}) = O(\log \frac{1}{p_x})$, where p_x is the access probability of x.

If we know the access frequencies in advance, we can also build a static binary search tree which gives the optimal expected access time for these frequencies, and one can show that the splay tree performance is within a constant factor of the optimal static binary search tree. This property is known as *static optimality* of splay trees, and there are several similar properties.

However, one question remains. What if we compare splay trees not against the optimum static search tree, but against the optimum dynamic search tree? That is, we assume that there is an adversary who knows the entire access sequence in advance and can perform arbitrary rotations on his search tree in anticipation of the future. How do splay trees compare to such an adversary? It is conjectured that even then splay trees are within a constant factor of the optimum. This is known as the *dynamic optimality conjecture*. For example, if we consider the access sequence $1, 2, 3, \ldots, n, 1, 2, \ldots, n, 1, \ldots$, a clever adversary only incurs a total cost of $O(m)$ (counting rotations and walks in the tree). It can be shown that splay trees also take $O(m)$ time, but the proof is very complicated. There are data structures which provably achieve total cost which is within a factor of $O(\log \log n)$ of the optimum cost.

6 Heaps/Priority Queues

In the next two lectures, we are going to consider heap data structures. The problem is as follows: We want to maintain a set of priorities, and support the operations insert, delete, findMin and deleteMin. One way of doing this is to use the classical binary heap, which stores a complete heap-ordered tree in an array and performs all the operations in $O(\log n)$ worst case time. We can generalize them to k-ary heaps, achieving a trade-off between insertion and deletion times. In the next two classes, we will consider more sophisticated heap structures.