COS 423: Theory of Algorithms Spring 2007

Lecture 2 — February 7, 2007

Prof. Robert Tarjan *Scribe: Wolfgang Mulzer*

1 Overview

In this lecture, we will discuss deletion in red black trees and give an analysis of the running time of the different red black tree operations. Furthermore, we will discuss the analysis of randomly constructed binary search trees, randomized quick-sort and treaps.

2 Deletion in Red Black Trees

In the last lecture, we saw how to insert elements into red black trees. Let us now look at deletion: In order to delete a node in a red black tree, we perform the standard deletion algorithm for binary search trees. If we end up deleting a red node, no more work is required. If we end up deleting a black node, we need to restore the red black tree invariant. When a black node is deleted, we think of the corresponding subtree as being "short of blackness", and we call its root a "short node". The restructuring proceeds by pushing the shortness up the tree until we reach a red node or the root. To do this, we perform recolorings or rotations, depending on the color of the sibling of the short node. Recolorings push the shortness up the tree, while rotations remove it. For a detailed case analysis, refer to the handout of the last lecture.

3 Analysis of Red Black Tree Operations

Since the red black tree invariant guarantees that the height of the tree is $O(\log n)$, all insertions and deletions take $O(\log n)$ time. Since rotations always lead to a terminal case, both insertions and deletions take a constant number of rotations (insertions at most 2, deletions at most 3). Recolorings, on the other hand, might go up all the way, so in the worst case, we need $O(\log n)$ recolorings in an insertions or deletion step.

However, we will now show that the number of recolorings is not too bad, in the following sense. Assume that we start with an empty tree and perform a sequence of m insertions and deletions. Then there exists a constant c such that the total number of recolorings during this sequence is at most cm. To prove this, we assign a potential Φ to a red black tree as follows:

$$\Phi = 2 \cdot \#\text{ of black nodes with 2 red children} + \#\text{ of black nodes with 2 black children}$$

We observe that the potential is initially zero and always positive. An inspection of the cases yields that the potential decreases by at least one on each recoloring, and that each insertion or deletion increases the potential by at most a constant c. Hence, the total number of recolorings can be at most cm.

4 More Comments on Red Black Trees

The red black trees presented in class are just one example. There exist many other ways to define colored balanced search trees.

If we merge red nodes with their black parents, we end up with a tree whose nodes have degree 2, 3 or 4, such that each leaf has the same depth. These trees are called 2,3,4-trees and can be generalized to B-trees, another example of a balanced tree data structure.

5 Average-Case Analysis for BSTs, Quicksort, and Treaps

We saw that standard binary search trees can degenerate arbitrarily badly when no restructuring is used. But what happens if we insert the nodes in a random order? In this case, we end up with a BST whose average path length is $O(\log n)$. The analysis turns out to be the same as the analysis of quicksort on a random permutation.

Reminder: Quicksort sorts a sequence of n elements by picking an arbitrary element as a pivot, partitioning on this element and recursing on the two sub-arrays. If we assume that the initial permutation is random, then a random element is picked as the pivot in each step. If we imagine that the pivots are inserted into a binary search tree in the order in which they are chosen by quicksort, the comparisons performed by the insertion process are the same as the comparisons performed by quicksort.

To analyze the expected number of comparisons, we have two options: Firstly, we could just set up a recurrence and solve it. The second method is more elegant. For two items i and j, let p_{ij} be the probability that quicksort compares them. This happens exactly when i or j is selected as a pivot before any other element in the interval $[i, j]$ is chosen. The probability for this event is $2/(j - i + 1)$. Thus, the expected number of comparisons is

$$\begin{aligned}
\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} p_{ij} &= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1} \\
&= \sum_{i=1}^{n-1} \sum_{k=2}^{n-i+1} \frac{2}{k} \\
&\leq \sum_{i=1}^{n-1} \sum_{k=1}^{n} \frac{2}{k} \\
&= (n-1) \sum_{k=2}^{n} \frac{2}{k} \\
&\approx (n-1) \ln n - 4.2(n-1)
\end{aligned}$$

Hence, the average running time of quicksort is $O(n \log n)$ and the average path length in a randomly constructed binary search tree is $O(\log n)$.

A problem with this kind of analysis is that there still are natural inputs for which the algorithm can perform very poorly (e.g. a sorted sequence for quicksort). A way to get around this for

quicksort is to put the randomization into the algorithm and to permute the elements to be sorted randomly in the beginning. A similar idea can also be used for binary search trees, which leads us to a data structure called *treap*.

In a treap, each node stores a key and a priority. The treap is a binary search tree with respect to the keys and a heap with respect to the priorities. The treap corresponding to a given set of key-priority pairs is unique. Insertions into treaps are performed like in standard binary search trees, followed by a sequence of rotations that restore the heap property. To delete an element, we rotate it down to the bottom of the tree and then remove it. The interesting property of treaps is that if the priorities are chosen uniformly at random from the interval $[0, 1]$, then all the treap operations take expected time $O(\log n)$.