423: Theory of Algorithms Spring 2007

Lecture 1 — February 7, 2007

Scribe:

1 Overview

In this lecture we will discuss Search trees, more specifically binary trees, balanced binary trees, and red black trees.

2 Problem

Suppose we are given a list of elements that consists of an id and a value. How can we efficiently find the value given the id? For an example, we are given a list of food items and their prices (ie apple: 50 cents)

We could step through the list until we find the element whose id matches. Then return the value. This brute force method, however, is expensive. Given N elements, it would take $O(N)$ time to find the element with the given id.

How can we do this more efficiently? Hash Tables!!!
The Good: The lookup time for a hash table is constant time, all we have to do is run a hash function over the given id.
The Bad: There can be collisions, especially if the number of elements is much larger than the hash table size. Also, hash tables loses any ordering information associated with the elements. The hashtable will be unable to efficiently answer questions such as *what is the cost for all food items that with g?*.

3 Binary Trees

Binary Trees perserve the order associated with a set of elements. Insertions, deletions, and accesses on a well balanced tree takes $O(logn)$ time.

Finds: To find an element with a given id, we first start at the root of the tree. Recursively we check to see if the given id is less than or greater than the current node's id. If its less, go left. If its more, go right.

Insertion: Do the the same comparison process as a find until you get to the bottom of the tree. And the new item as a leaf node.

Deletions: When deleting a leaf node, just remove it. When deleting a node with one child, replace the node with its child and then delete it. Deletions for 2 nodes are trickier. If we want

to remove node n, we first replace n with a node that's easier to replace (ie one of the above 2 instances) and then delete it. To find the element to replace n wtih, we just go to n's left child and then follow the right path all the way down. Or, we go to n's right and then go down the left branch to find the item to replace.

For maximum efficiency, we want to minimize the tree depth. However, this is not gaurenteed. In the worse case scenario, we end up with a tree thats one long path.

4 Balanced Trees

We want a guarentee of the good behavior we get from a balanced binary tree. *AVL-trees* are *height balanced trees*. For any node, the length of the right path and the length of the left path differs only by 1. This is enough to guarentee the depth is logrithmic. *weight balanced trees*: For any node, its two subtrees have the same number of nodes by a constant factor.

5 Red-Black Trees

Every node is labeled either red or black. The rules are:

1. Black: evey path from top to bottom has the same number of black nodes
2. Red: any red node has a black parent

Claim 1. *the depth is less than* $2lgn$

Let b be the number of black nodes. There are at most b red node since every red node has a black parent. THus there are no more than $2b$ nodes.

To insert an element, we first store it in a red node. Then insert the node in the same manner as the binary tree. Afterwards we need to fixi any color rules that are broken. This requires rotations based on the structure and labels of the nodes. For more information on insertion, please see the handout.