Illinois Institute of Technology
Department of Computer Science

Lecture 9: February 10, 2016

CS 430 Introduction to Algorithms
Spring Semester, 2016

Augmenting Balanced Trees

For particular applications, it is useful to modify a standard data structure to support additional functionality. Two examples are presented here, order-statistic trees and interval trees.

Order-Statistic Trees

In Lecture 5 (January 27) we discussed the selection problem: given an array, find the kth largest element. Can we modify red-black trees to allow us to solve this problem? Indeed we can, with order-statistic trees (section 14.1 of CLRS).

We *augment* the each node in the red-black tree with its size, where $size(x)$ is defined as the number of items in the subtree rooted at x. With this new information, finding the kth largest element of a tree is a simple process which can be performed in $O(height(T))$ time—and since red-black trees are of height $O(\log n)$, the running time of the selection algorithm is $O(\log n)$ (page 341 of CLRS).

This size field must be updated whenever the tree is modified. Recall that insertion into a red-black tree occurs in two phases. In the first phase, the new node is inserted as the child of an existing node, and sizes can be updated by adding 1 to all size values along the path from the new node to the root. Since the height of the tree is $O(\log n)$, this takes $O(\log n)$ time. In the second phase, rotations are performed in order to restore the red-black properties of the tree. Fortunately, rotations make only local changes to the nodes around the edge on which the rotation takes place. Since at most two rotations are required for an insertion, maintaining the size information takes $O(1)$ additional time. For similar reasons, updating the size field after a deletion also requires only $O(\log n)$ time.

Interval Trees

Interval trees (section 14.3 of CLRS) are red-black trees extended to support operations on intervals of real numbers. In particular, the data structure stores intervals and, when we search for an interval i in the tree, a pointer is returned to some interval in the tree that overlaps i, if one exists.

For each interval i in the tree we store its high endpoint, $high(i)$, and its low endpoint, $low(i)$. We use the low endpoint as the key, so that the tree supports efficient search for low endpoints. At each node i we also store $max(i)$, the maximum value of all the high endpoints in the subtree rooted at i.

Maintaining this supplementary information during insertions and deletions is not difficult. Clearly, $max(i) = \max\{high(i), max(left(i)), max(right(i))\}$, so the max field can be updated in constant time for each rotation performed. Therefore, insertion and deletion still run in $O(\log n)$ time.

To search for an interval i in the tree T, we compare $max(left(T))$ to $low(i)$. If $low(i)$ is larger, no overlapping interval could possibly exist in the left subtree of T, so we search recursively for i in $right(T)$. If $\max left(T)$

is larger, if there is a solution, there must be a solution in $left(T)$,[1] so we search recursively for i in $left(T)$. Searching in this manner takes $O(\log n)$ time.

[1]There may also be a solution in $right(T)$, or there may be no solution at all, but there cannot *only* be a solution in $right(T)$.