Illinois Institute of Technology
Department of Computer Science

# Lecture 7: September 15, 2010

CS 430 Introduction to Algorithms
Fall Semester, 2010

## 1 Binary search trees

Sorted arrays are attractive that we can use binary search to locate any element in $\Theta(\log n)$ time. Unfortunately, this comes at a significant cost: insertion or deletion of an element requires $\Theta(n)$ time. With a linked list, locating an element requires $\Theta(n)$ time, but insertion or deletion requires only $\Theta(1)$ time.

We can use trees to combine these benefits. Each node in a binary tree contains a record (our examples here assume that the record is a number) and has pointers to two children, one left child and one right child. In particular, a *binary search tree* is *lexicographic*; that is, the values in the left subtree are all less than the value at the root and the values in the right subtree are all greater than the value at the root.

### 1.1 Basic operations

#### 1.1.1 Search

Searching for an element $x$ in a binary search tree is simple: If the tree is empty, then $x$ is not in the tree. If the tree is not null, say the element stored at the root is $r$. If $x = r$, the root contains the element and the search is complete. If $x < r$ then $x$ must be in the left subtree if it is in the tree at all. Thus, we recursively search for $x$ in the left subtree. Likewise, if $x > r$, we recursively search of $x$ in the right subtree.

If the tree is "nicely balanced," this runs in $\Theta(\log n)$ time. If the tree is badly skewed it takes $\Theta(n)$ time, no better than a linked list. In general, it runs in time $\Theta(\text{HEIGHT}(T))$, where $\text{HEIGHT}(T)$ is the height of tree $T$. We will return to this analysis later.

#### 1.1.2 Insertion

To insert an element $x$, first we search for it. If it already exists in the tree, we do not need to insert it. If it is not in the tree, we replace the null pointer reached by the search routine with a pointer to a new record containing $x$ and null child pointers. The time analysis is identical to that of unsuccessful search.

#### 1.1.3 Finding the maximum or minimum element

Finding the maximum element in a binary search tree is again trivial; successively follow right pointers until a null pointer is reached. The node with the null pointer contains the maximum element of the tree. Like search and insertion, this takes time $\Theta(\text{HEIGHT}(T))$.

Finding the minimum is the symmetric operation.

#### 1.1.4 Finding the successor to an element

To find the successor to an element $x$, first we find $x$. If $x$'s right pointer is not null, we find the minimum of the right subtree. Otherwise we repeatedly follow parent pointers until the pointer we follow is a *left* child pointer. If we arrive at the root before reaching such a node, $x$ is the maximum and has no successor. This also takes time $\Theta(\text{HEIGHT}(T))$.

#### 1.1.5 Deletion

Deletion of an element $z$ from a binary search tree is not as simple as the other operations and divides into the following cases:[1]

- If $z$ has no children (i.e., both of its child pointers are null), it can be simply removed.
- If $z$ has one child, we remove $z$ and its parent "adopts" the child.
- If $z$ has two children, we find its successor $y$, which must have a null left pointer. We place the contents of $y$ in place of $z$ and delete the "old" $y$—since its left pointer is null, it has no more than one child and we already know how to delete it.

Deletion is *asymmetrical* as it works by finding the successor rather than the predecessor. A tree that has undergone a deletion is *biased* in some fashion. (The algorithm could use predecessors instead of successors, but that would only bias the tree the other way.)

### 1.2 Behavior of a binary search tree

We have shown that the basic operations on a binary search tree all run in time $\Theta(\text{HEIGHT}(n))$. If a tree is grown with random insertions and no deletions,[2] what can we say about its expected height? It turns out that the expected height of such a tree is $c \lg n$, where $c > 2$ satisfies $c \ln \frac{2e}{c} = 1$, so $c \approx 4.31107$. We do not prove this here.[3]

Instead, we examine the average search time of such a tree. Define the external path length of a tree as

$$\text{EPL}(T) = \sum_{l \text{ is a leaf of } T} \text{DEPTH}(l)$$

That is, the external path length of a tree is the sum of the depths of each leaf. Then the *average unsuccessful search time* of tree $T$ is $\frac{\text{EPL}(T)}{n+1}$, since a binary tree with $n$ nodes has $n+1$ leaves.

We have

$$\begin{aligned} \text{EPL}(\square) &= 0 \\ \text{EPL}(T) &= \text{EPL}(T_l) + \text{EPL}(T_r) + n + 1 \end{aligned}$$

where $T_l$ and $T_r$ are the left and right subtrees, respectively, of $T$.

[1] Figure 12.4 on page 297 of CLRS demonstrates these three operations.

[2] Recall the asymmetry in deletions. There have been no thorough analyses of trees grown with random insertions and deletions.

[3] Section 12.4 of CLRS, which you do not need to read, performs a very crude approximation to this calculation. It is more thoroughly worked out in L. Devroye, "A note on the expected height of binary search trees," *Journal of the Association of Computing Machinery* 33 (1986), pp. 489-498.

The internal path length is defined similarly:

$$\mathrm{IPL}(T) = \sum_{x \text{ is an internal node of } T} \textsc{depth}(x)$$

That is, the internal path length of a tree is the sum of the depths of the internal nodes.

This leads to the inductive definition

$$\begin{aligned} \mathrm{IPL}(\square) &= 0 \\ \mathrm{IPL}(T) &= \mathrm{IPL}(T_l) + \mathrm{IPL}(T_r) + n - 1 \end{aligned}$$

Since induction shows $\mathrm{EPL}(T) = \mathrm{IPL}(T) + 2n$, studying the external path length tells us about the internal path length as well.

### 1.2.1 Average-case unsuccessful search time

Let us denote the expected value of $\mathrm{EPL}(T)$, where $T$ has $n+1$ leaves (i.e. $n$ internal nodes), by $E_n$. Clearly $E_0 = 0$. For larger trees,

$$\begin{aligned} E_n &= \sum_{k=1}^{n} (E_{k-1} + E_{n-k} + n + 1) \cdot \mathbf{Pr}\{k \text{ is the root of the tree}\} \\ &= \frac{1}{n} \sum_{k=1}^{n} (E_{k-1} + E_{n-k} + n + 1) \\ &= n + 1 + \frac{2}{n} \sum_{k=0}^{n-1} E_k \end{aligned}$$

But this is the same recurrence we saw in our analysis of quicksort, with the solution $E_n = 2nH_n$. So the average unsuccessful search time is

$$\begin{aligned} \frac{E_n}{n+1} &\approx \frac{E_n}{n} \\ &\approx 2H_n \\ &\approx 1.38 \lg n \\ &= \Theta(\log n) \end{aligned}$$

### 1.2.2 Worst-case unsuccessful search time

In the worst case, the tree is entirely unbalanced—it simply mimics a list.[4] This gives us

$$E_n = E_{n-1} + n + 1$$

where $E_n$ is now the worst-case value of $\mathrm{EPL}(T)$. This can be solved to yield

$$E_n = \frac{(n+1)(n+2)}{2}$$

So

$$\begin{aligned} \frac{E_n}{n+1} &\approx \frac{n}{2} \\ &= \Theta(n) \end{aligned}$$

[4] This can be proven using techniques similar to those we used to find the worst case of quicksort.

### 1.2.3 Best-case unsuccessful search time

The best case is a perfectly balanced tree, giving

$$E_n = E_{\lfloor \frac{n}{2} \rfloor} + E_{\lceil \frac{n}{2} \rceil} + n + 1$$

We have already solved this in our analysis of quicksort:

$$\begin{aligned} E_n &\approx 2E_{\frac{n}{2}} + n \\ &= \Theta(n \lg n) \end{aligned}$$

So

$$\frac{E_n}{n+1} = \Theta(\lg n)$$