CMSC 351 - Introduction to Algorithms
Spring 2012
Lecture 14

Instructor: MohammadTaghi Hajiaghayi
Scribe: Rajesh Chitnis

1 Introduction

In this lecture we will look at Trees and Binary Search Trees.

2 Binary Search Trees

We see **implicit** representations of trees for heaps. For explicit representation all nodes have $m+1$ pointers, one to the parent and m to the children, where m is the maximal number of children in a tree. Alternatively, if m is unknown, we have two pointers: one to the first child and second to the next sibling (a linked list or doubly-linked list). Binary search trees is another way to implement **dictionary operations** without randomization and it is comparison-based (the bounds are the worst-case bounds). The dictionary operations are as follows:

1. Search(x): Find x or determine x is not there (assume all elements are distinct for now).
2. Insert(x): Insert x unless it is already there.
3. Delete(x): Delete x if it is there.

Binary search trees also implement more complicated operations efficiently. We use explicit representation for binary trees by keeping a record containing at least three fields: key, left and right (child). Due to explicit representation, any node may be added or removed unlike heaps where we only remove or insert leaves. Each key in the binary search tree serves to divide the range of the keys below. The keys in the left subtree are all smaller than it and the keys in the right subtree are all greater than it. We say a tree is **consistent** if all keys satisfy this condition.

Search Operation: We first compare x against the root of the tree, whose value is say r. If $x = r$, then we are done. If $x < r$, we continue from the left child. Otherwise we continue from the right child.

Insertion Operation: First we search for x and abort insert if x is already in the tree. Otherwise, the search ends at a leaf and the new key can then be inserted below that leaf depending on the value of x (at left or right). Note that the tree remains consistent.

Deletion Operation: Deletion is more complicated. It is easy if it is a leaf (by making it NIL). Even if the node has one child, it is easy by changing the child pointer of the parent to the child of x (instead of x). What about if the node x has two children? First we find the predecessor of x in the tree which is a node at least as large as all the keys in the left subtree of x and smaller than all the keys in the right subtree of x. To find it first we go to the left child of x and then follow the right child as long as we can. When we end up by finding a vertex say p, since it has only at most left child, we will delete it easily and replace x with p. The consistency is preserved since p is the largest element of the left subtree of x, i.e., the tree rooted at the left child of x.

Time Complexity: All running times depend on the shape of the tree and the location of the relevant node; in the worst case it is $O(\text{height}(T))$ where the height (or depth) of a tree is defined as the maximum length of a path from its root to any leaf. Thus a rooted tree with only one vertex (the root) has height 0 and the height of an empty tree is defined to be -1. If the tree is balanced, i.e., $\text{height}(T) = O(\log n)$, where n is the number of elements, all operations are in $O(\log n)$ and thus efficient. If the keys are inserted in a random order, then one can probe that the expected height of the tree is $O(\log n)$ (like the depth of quicksort) and thus dictionary operations are efficient in the average case. However if the insertions are in a sorted, or closed to sorted, order, then the tree would be a linked list (a long path) and the operations can take $\Omega(n)$ time. To prevent this worst case we can use **AVL** trees.

3 AVL Trees

Definition 1 *AVL trees are binary search trees such that for every node, the difference between the heights of its left and right subtrees is at most one.*

Theorem 1 *The height h of an AVL tree on n vertices is $O(\log n)$.*

The idea of AVL trees is that whenever the property above is violated, we rebalance the tree by performing a **rotation** operation. See details in the book [1]. All operations in AVL trees take $O(\text{height}) = O(\log n)$. Empirical studies have shown that the average search time to be approximately $\log_2 n + 0.25$ comparisons. Note that we can perform delete-max() or delete-min() operations in AVL trees by following right of left children until we reach a leaf. Again the running time is $O(\text{height}) = O(\log n)$. Now we can do a sort using binary search trees or AVL trees as follows:

1. First insert all keys in the tree in the order of the input.
2. Do n delete-min() operations and insert the retained elements in the output array.

Both steps and thus the whole sort is in $O(n \log n)$ and so it is efficient. The sort here is a comparison-based sort, and is not an in-place sort.

References

[1] Udi Manber, *Introduction to Algorithms - A Creative Approach*