Lecture 8: More on Binary Search Trees

Read: Chapt 4 of Weiss, through 4.4.

Deletion: We continue our discussion of binary search trees, and consider how to delete an element from the tree. Deletion is a more involed than insertion. There are a couple of cases to consider. If the node is a leaf, it can just be deleted with no problem. If it has no left child (or equivalently no right child) then we can just replace the node with it's left (or right) child.

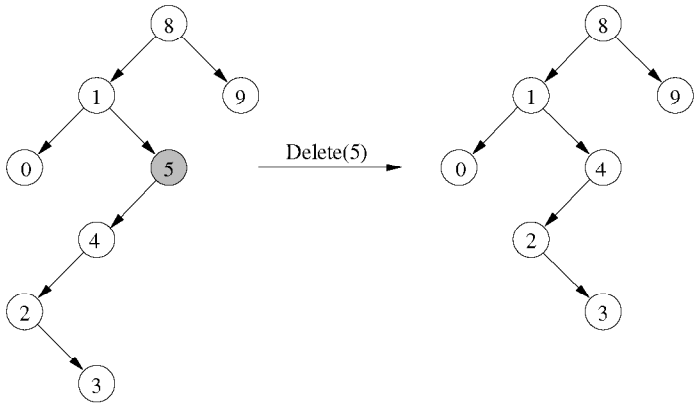

Figure 19: Binary tree deletion: Case of one child.

If both children are present then things are trickier. Removal of the node would create a "hole" in the tree, and we need to fill this hole. The idea is to fill the hole with the element either immediately preceding or immediately following the deleted element. Because the tree is ordered according to an inorder traversal, the candidate elements are the inorder predecessor or successor to this node. We will delete, say, the inorder successor, and make this the replacement. You might ask, what if the inorder successor has two children? It turns out this cannot happen. (You should convince yourself of this. Note that because this node has two children, the inorder successor will be the leftmost node in its right subtree.)

Before giving the code we first give a utility function, `findMin()`, which returns a pointer to the element with the minimum key value in a tree. This will be used to find the inorder successor. This is found by just following left links as far as possible. We assume that the argument p is non-null (which will be the case as we shall see later). (Also note that variables are being passed by value, so modifying p will have no effect on the actual argument.) As with the insertion method, the initial call is made to the root of the tree, `delete(x, root)`. Also, as with insertion, rather than using a parent link to reach up and modify the parent's link, we return the new result, and the parent stores this value in the appropriate link.

Binary Tree Deletion

```
BinaryNode findMin(BinaryNode p) {                  // assume (p != null)
    while (p.left != null) p = p.left;
    return p;
}

BinaryNode delete(Element x, BinaryNode p) {
    if (p == null)                                  // fell out of tree?
```

[1]Copyright, David M. Mount, 2001

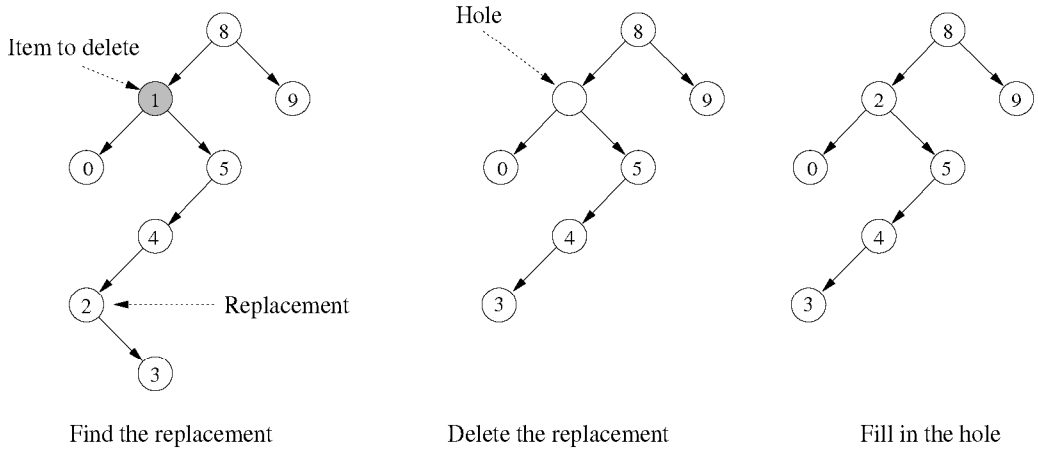

Figure 20: Binary tree deletion: Case of two children.

```
        ... Error: deletion of nonexistent element!...
    else {
        if (x < p.data)                              // x in left subtree
            p.left = delete(x, p.left);
        else if (x > p.data)                         // x in right subtree
            p.right = delete(x, p.right);
                                                     // x here, either child empty?
        else if (p.left == null || p.right == null) {
            BinaryNode repl;                         // get replacement
            if (p.left == null) repl = p.right;
            if (p.right == null) repl = p.left;
            return repl;
        }
        else {                                       // both children present
            p.data = findMin(p.right).data;          // copy replacement
            p.right = delete(p.data, p.right);       // now delete the replacement
        }
    }
    return p;
}
```

In typical Java fashion, we did not worry about deleting nodes. Where would this be added to the above code? Notice that we did not have a special case for handling leaves. Can you see why this is correct as it is?

Analysis of Binary Search Trees: It is not hard to see that all of the procedures `find()`, `insert()`, and `delete()` run in time that is proportional to the height of the tree being considered. (The `delete()` procedure is the only one for which this is not obvious. Note that each recursive call moves us lower in the tree. Also note that `findMin()`, is only called once, since the node that results from this operation does not have a left child, and hence will itself not generate another call to findMin.)

The question is, given a binary search tree T containing n keys, what is the height of the tree? It is not hard to see that in the worst case, if we insert keys in either strictly increasing or strictly decreasing order, then the resulting tree will be completely degenerate, and have

height $n-1$. On the other hand, following the analogy from binary search, if the first key to be inserted is the median element of the set of keys, then it will nicely break the set of keys into two sets of sizes roughly $n/2$ each. This will result in a nicely balanced tree, whose height will be $O(\log n)$. Thus, if you had a million nodes, the worst case height is a million and the best is around 20, which is quite a difference. An interesting question is what is the expected height of the tree, assuming that keys are inserted in random order.

Our textbook gives a careful analysis of the average case, but it is based on solving a complex recurrence. Instead, we will present a simple probabilistic analysis to show that the number of nodes along the leftmost chain of the tree is expected to be $\ln n$. Since the leftmost path is a representative path in the tree, it is not surprising the that average path length is still $O(\log n)$ (and a detailed analysis shows that the expected path length is actually $2 \ln n$).

The proof is based on the following observation. Consider a fixed set of key values and suppose that these keys are inserted in random order. One way to visualize the tree is in terms of the graphical structure shown in the following figure. As each new node is added, it locates the current interval containing it, and it splits a given interval into two subintervals. As we insert keys, when is it that we add a new node onto the left chain? This happens when the current key is smaller than all the other keys that we have seen up to now. For example, for the insertion order $\langle 9, 3, 10, 6, 1, 8, 5\rangle$ the minimum changes three times, when 9, 3, and 1 are inserted. The corresponding tree has three nodes along the leftmost chain (9, 3, and 1). For the insertion order $\langle 8, 9, 5, 10, 3, 6, 1\rangle$ the minimum changes four times, when 8, 5, 3, and 1 are inserted. The leftmost chain has four nodes (8, 5, 3, and 1).

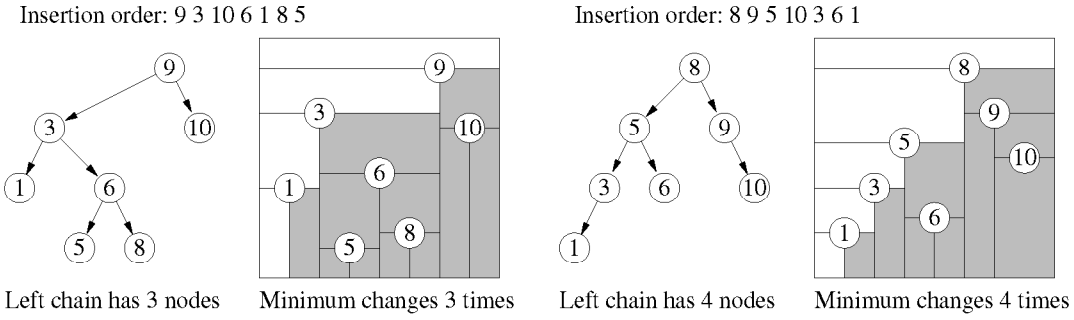

Figure 21: Length of the leftmost chain.

So this suggests the following probabilistic question: As you scan a randomly permuted list of distinct numbers from left to right, how often do you expect the minimum value to change? Let p_i denote the probability that the minimum changes when the ith key is being considered. What is the probability that the ith element is a new minimum? For this to happen, this key must be the smallest among all the first i keys. We assert that the probability that this happens is $1/i$. If you consider the first i keys, exactly one of them is the minimum among these keys. Since we are considering random permutations, this minimum is equally likely to be in any of the i positions. Thus, there is a $1/i$ chance that it is in the last (ith) position.

So with probability $p_i = 1/i$ the minimum changes when we consider the ith key. Let x_i be a random indicator variable which is 1 if the minimum changes with the ith key and 0 otherwise. The total number of minimum changes is the random variable $X = x_1 + x_2 + \ldots + x_n$. We have $x_i = 1$ with probability p_i, and $x_i = 0$ with probability $1 - p_i$. Thus the expected number

of changes is

$$E(X) \;=\; \sum_{i=1}^{n}(1 \cdot p_i + 0 \cdot (1 - p_i)) \;=\; \sum_{i=1}^{n} p_i \;=\; \sum_{i=1}^{n} \frac{1}{i} \;=\; \ln n + O(1).$$

The last line follows from the fact that the sum $(1/i)$ is the Harmonic series, which we saw in an earlier lecture. This completes the proof that the expected length of the leftmost chain is roughly $\ln n$.

Interestingly this analysis breaks down if we are doing deletions. It can be shown that if we alternate random insertions and random deletions (keeping the size of the tree steady around n), then the height of the tree will settle down at $O(\sqrt{n})$, which is worse that $O(\log n)$. The reason has to do with the fact that the replacement element was chosen in a skew manner (always taking the minimum from the right subtree). Over the course of many deletions, this can result in a tree that is left-heavy. This can be fixed by alternating taking the replacement from the right subtree with the left subtree resulting in a tree with expected height $O(\log n)$.