Lecture 10: Splay Trees

Read: Chapt 4 of Weiss, through 4.5.

Splay Trees and Amortization: Recall that we have discussed binary trees, which have the nice property that if keys are inserted and deleted randomly, then the expected times for insert, delete, and member are $O(\log n)$. Because worst case scenarios can lead $O(n)$ behavior per operation, we were lead to the idea of the height balanced tree, or AVL tree, which guarantees $O(\log n)$ time per operation because it maintains a balanced tree at all times. The basic operations that AVL trees use to maintain balance are called rotations (either single or double). The disadvantages of AVL trees are that we need to maintain balance information in the nodes, and the routines for updating the AVL tree are somewhat more complicated that one might generally like.

Today we will introduce a new data structure, called a *splay tree*. Like the AVL tree, a splay tree is a binary tree, and we will use rotation operations in modifying the tree. Unlike an AVL tree no balance information needs to be stored. Because a splay tree has no balance information, it is possible to create unbalanced splay trees. Splay trees have an interesting *self-adjusting* nature to them. In particular, whenever the tree becomes unbalanced, accesses to unbalanced portions of the tree will naturally tend to balance themselves out. This is really quite clever, when you consider the fact that the tree has no idea whether it is balanced or not! Thus, like an unbalanced binary tree, it is possible that a single access operation could take as long as $O(n)$ time (and not the $O(\log n)$ that we would like to see). However, the nice property that splay trees have is the following:

Splay Tree Amortized Performance Bound: Starting with an empty tree, the total time needed to perform any sequence of m insertion/deletion/find operations on a splay tree is $O(m \log n)$, where n is the maximum number of nodes in the tree.

Thus, although any one operation may be quite costly, over any sequence of operations there must be a large number of efficient operations to balance out the few costly ones. In other words, over the sequence of m operations, the average cost of an operation is $O(\log n)$.

This idea of arguing about a series of operations may seem a little odd at first. Note that this is not the same as the average case analysis done for unbalanced binary trees. In that case, the average was over the possible insertion orders, which an adversary could choose to make arbitrarily bad. In this case, an adversary could pick the worst sequence imaginable, and it would still be the case that the time to execute the entire sequence is $O(m \log n)$. Thus, unlike the case of the unbalanced binary tree, where the adversary can force bad behavior time after time, in splay trees, the adversary can force bad behavior only once in a while. The rest of the time, the operations execute quite efficiently. Observe that in many computational problems, the user is only interested in executing an algorithm once. However, with data structures, operations are typically performed over and over again, and we are more interested in the overall running time of the algorithm than we are in the time of a single operation.

This type of analysis based on sequences of operations is called an *amortized analysis*. In the business world, amortization refers to the process of paying off a large payment over time in small installments. Here we are paying for the total running time of the data structure's algorithms over a sequence of operations in small installments (of $O(\log n)$ each) even though each individual operation may cost much more. Amortized analyses are extremely important in data structure theory, because it is often the case that if one is willing to give up the requirement that every access be efficient, it is often possible to design data structures that are simpler than ones that must perform well for every operation.

Splay trees are potentially even better than standard search trees in one sense. They tend to bring recently accessed data to up near the root, so over time, we may need to search less than $O(\log n)$ time to find frequently accessed elements.

Splaying: As we mentioned earlier, the key idea behind splay trees is that of self-organization. Imagine that over a series of insertions and deletions, our tree has become rather unbalanced. If it is possible to repeatedly access the unbalanced portion of the tree, then we are doomed to poor performance. However, if we can perform an operation that takes unbalanced regions of the tree, and makes them more balanced then that operation is of interest to us. As we said before, since splay trees contain no balance information, we cannot selectively apply this operation at positions of known imbalance. Rather we will perform it everywhere along the access path to every node. This basic operation is called *splaying*. The word splaying means "spreading", and the splay operation has a tendency to "mix up" trees and make them more random. As we know, random binary trees tend towards $O(\log n)$ height, and this is why splay trees seem to work as they do.

Basically all operations in splay trees begin with a call to a function `splay(x,t)` which will reorganize the subtree rooted at node t, bringing the node with key value x to the root of the tree, and generally reorganizing the tree along the way. If x is not in the tree, either the node immediately preceding or following x will be brought to the root.

Here is how `splay(x,t)` works. We perform the normal binary search descent to find the node v with key value x. If x is not in the tree, then let v be the last node visited before we fall out of the tree. If v is the root then we are done. If v is a child of the root, then we perform a single rotation (just as we did in AVL trees) at the root to bring v up to the root's position. Otherwise, if v is at least two levels deep in the tree, we perform one of four possible double rotations from v's grandparent. In each case the double rotations will have the effect of pulling v up two levels in the tree. We then go up to the new grandparent and repeat the operation. Eventually v will be carried to the top of the tree. In general there are many ways to rotate a node to the root of a tree, but the choice of rotations used in splay trees is very important to their efficiency.

The rotations are selected as follows. Recall that v is the node containing x (or its immediate predecessor or successor). Let p denote x's parent and let g denote x's grandparent. There are four possible cases for rotations. If x is the left child of a right child, or the right child of a left child, then we call this a *zig-zag* case. In this case we perform a double rotation to bring x up to the top. See the figure below. Note that this can be accomplished by performing one single rotation at p and a second at g.

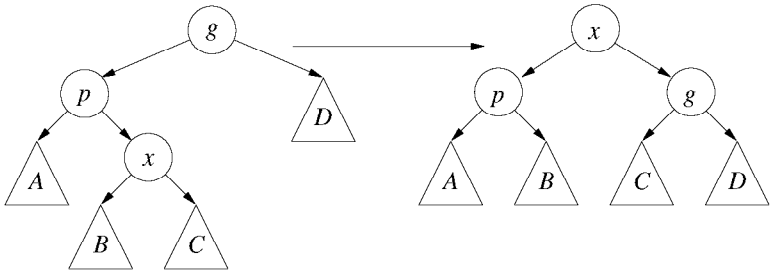

Figure 27: Zig-zag case.

Otherwise, if x is the left child of a left child or the right child of a right child we call this a *zig-zig* case. In this case we perform a new type of double rotation, by first rotating at g and then rotating at p. The result is shown in the figure below.

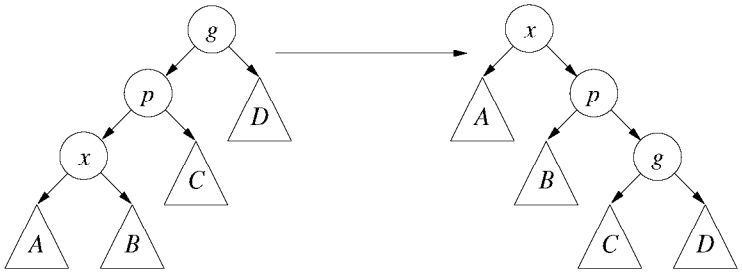

Figure 28: Zig-zig case.

A complete example of `splay(3,t)` is shown in the next figure.

Splay Tree Operations: Let us suppose that we have implemented the splay operation. How can we use this operation to help us perform the basic dictionary operations of insert, delete, and find?

To find key x, we simply call `splay(x,t)`. If x is in the tree it will be transported to the root. (This is nice, because in many situations there are a small number of nodes that are repeatedly being accessed. This operation brings the object to the root so the subsequent accesses will be even faster. Note that the other data structures we have seen, repeated find's do nothing to alter the tree's structure.)

Insertion of x operates by first calling `splay(x,t)`. If x is already in the tree, it will be transported to the root, and we can take appropriate action (e.g. error message). Otherwise, the root will consist of some key w that is either the key immediately before x or immediately after x in the set of keys. Let us suppose that it is the former case ($w < x$). Let R be the right subtree of the root. We know that all the nodes in R are greater than x so we make a new root node with x as data value, and make R its right subtree. The remaining nodes are hung off as the left subtree of this node. See the figure below.

Finally to delete a node x, we execute `splay(x,t)` to bring the deleted node to the root. If it is not the root we can take appropriate error action. Let L and R be the left and right subtrees of the resulting tree. If L is empty, then x is the smallest key in the tree. We can delete x by setting the root to the right subtree R, and deleting the node containing x. Otherwise, we perform `splay(x,L)` to form a tree L'. Since all the nodes in this subtree are already less than x, this will bring the predecessor w of x (i.e. it will bring the largest key in L to the root of L, and since all keys in L are less than x, this will be the immediate predecessor of x). Since this is the largest value in the subtree, it will have no right child. We then make R the right subtree of the root of L'. We discard the node containing x. See the figure below.

Why Splay Trees Work: As mentioned earlier, if you start with an empty tree, and perform any sequence of m splay tree operations (insert, delete, find, say), then the total running time will be $O(m \log n)$, where n is the maximum number of elements in the tree at any time. Thus the average time per operation is $O(\log n)$, as you would like. Proving this involves a complex *potential argument*, which we will skip. However it is possible to give some intuition as to why splay trees work.

It is possible to arrange things so that the tree has $O(n)$ height, and hence a splay applied to the lowest leaf of the tree will take $O(n)$ time. However, the process of splaying on this node will result in a tree in which the average node depth decreases by about half. (See the example on page 135 of Weiss.) To analyze a splay tree's performance, we model two quantities:

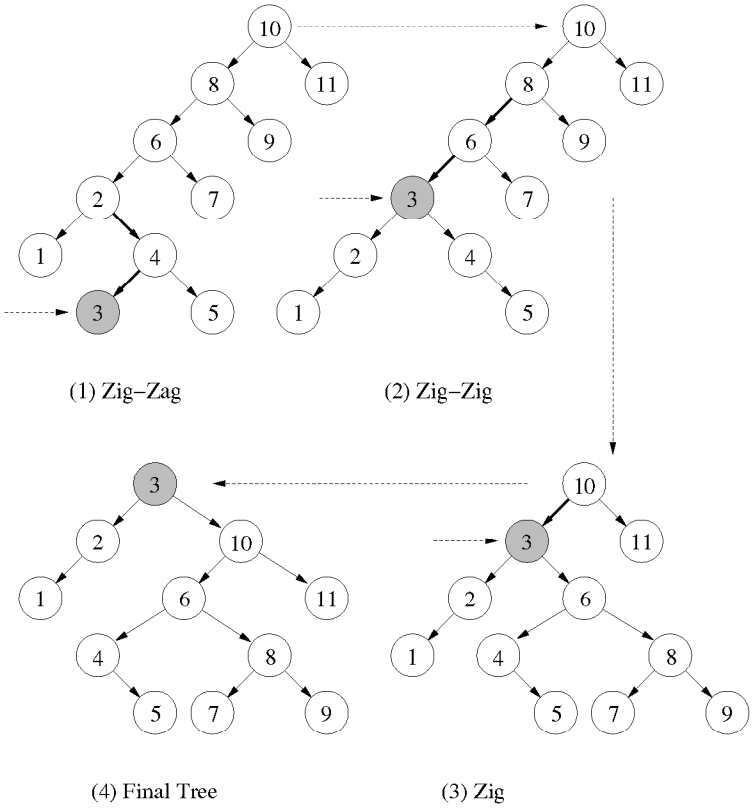

Figure 29: Example showing the result of Splay(3).

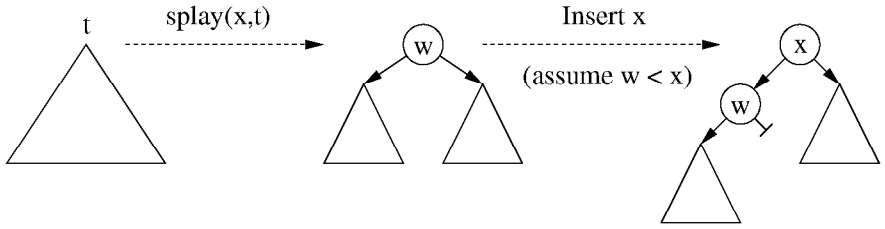

Figure 30: Insertion of x.

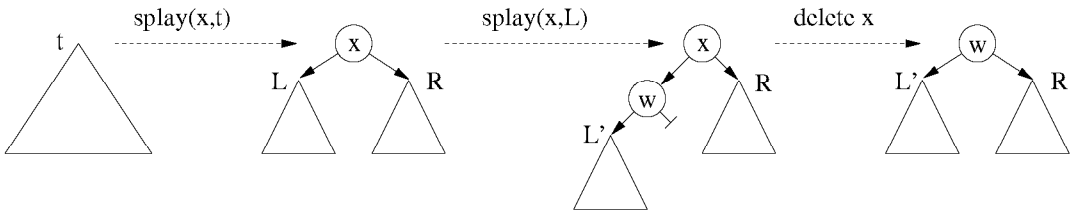

Figure 31: Deletion of x.

Real cost: the time that the splay takes (this is proportional to the depth of the node being splayed), and

Increase in balance: the extent to which the splay operation improves the balance of the tree.

The analysis shows that if the real cost is high, then the tree becomes much more balanced (and hence subsequent real costs will be lower). On the other hand, if the tree becomes less balanced, then the real cost will be low. So you win either way.

Consider, for example, the zig-zig case. Label the subtrees as shown in the zig-zig figure above. Notice that the element we were searching for is in the subtree rooted at x, and hence is either in A or B. Also note that after performing the rotation, depths of the in subtrees A and B have decreased (by two levels for A and one level for B). Consider the total number of elements in all the subtrees A, B, C and D.

A and B are light: If less than half of the elements lie in A and B, then we are happy, because as part of the initial search, when we passed through nodes g and p, we eliminated over half of the elements from consideration. If this were to happen at every node, then the total real search time would be $O(\log n)$.

A and B are heavy: On the other hand, if over half of the elements were in A and B, then the real cost may be much higher. However, in this case over half of these elements have decreased their depth by at least one level. If this were to happen at every level of the tree, the average depth would decrease considerably.

Thus, we are happy in either case: either because the real cost is low or the tree becomes much more balanced.