Chapter 12

Splay Trees

(Revision Date: 28 March 1999)

12.1 Introduction

We saw in Chapter 9 that the average binary search tree supports insertion, deletion, and membership operations in $O(\lg n)$ time but that there are worst case trees for which these operations are in $O(n)$. One approach to avoiding the worst case behavior is to require that the tree always remain balanced. The AVL tree of Chapter 10 is one such balanced tree. Balanced trees guarantee $O(\lg n)$ time per operation but do so at the cost of a high constant of proportionality. AVL trees adjust their structure after an insertion or deletion operation if necessary to maintain balance. They do not adjust their structure in response to a membership operation.

Splay trees are "self-adjusting" binary search trees in which the adjustment is done on the basis of the history of accesses rather than in an attempt to maintain an explicit balance. After access, a tree node is moved to the root by a sequence of rotations called "splaying." No balance information need be stored in the nodes and, in general, splay trees are not balanced. The most recently accessed nodes in a splay tree will tend to be close to the root. Accessing these nodes is less expensive than accessing nodes further down the tree. Because of data locality effects, having the most recently accessed nodes near or at the root can be a big gain.

The major point about splay trees is that the amortized performance of a sequence of m operations is $O(m \lg n)$. n is the total number of insertions (the maximum size attained by the tree during the sequence of m operations). Any individual operation in the sequence could be as costly as $O(n)$; there is no guarantee that a particular operation in the sequence will be efficient. Compare this with the performance of a sequence of operations in an AVL tree. Since each operation on the AVL tree is in $O(\lg n)$, the sequence of m operations will be in $O(m \lg n)$. Splay trees have the same asymptotic performance over a sequence of operations as do balanced trees. In the splay tree, some operations in the sequence may be inefficient, but other operations will be super-efficient to make up for them.

So what's a sequence of operations? It's any sequence of `Insert`, `Delete`, or `Search` operations performed in building and manipulating a splay tree from scratch. There are some constraints, of course. For example, the k-th deletion must have been preceded by at least k insertions.

12.2 Tree Operations

The `Insert` operation in a splay tree works by first doing a normal binary-search tree insertion, then splaying the inserted node. This puts the inserted node at the root. If the insertion fails because a duplicate element is already in the tree, the node holding the duplicate is splayed.

The `Search` operation searches the splay tree for the element sought. If found, that node is splayed to the root. If not found, the last node encountered on the search path (the parent of the non-existent element) is splayed to the root.

The `Delete` operation, as in regular binary-search trees, deletes the current element. In a splay tree, `Delete` involves three splaying operations. The current node is splayed to the root, producing a new splay tree T. Let T_L and T_R be the left and right subtrees of T. Disconnect these subtrees from T, then splay the largest element in T_L and the smallest element in T_R, after which T_L will have no right subtree and T_R will have no left subtree. Now, form a new tree T by making T_R the right subtree of T_L (or vice-versa, if T_L is empty). This new tree is the original tree after the deletion.

Another approach to `Delete` uses just one splay operation. The current node is splayed to the root, then normal `BSTree` deletion is performed on it. This certainly works for deletion, but may not satisfy the amortized performance requirement.

12.3 Overview of the Splaying Operation

A binary tree is *splayed* at a node `X` by traversing the tree from `X` to the root, performing sequences of single rotations along the way. The following splay operation is repetitively applied, beginning at node `X`, until `X` is at the root.

1. If `X` is root, do nothing.
2. If `X` has no grandparent (*i.e.*, the parent of `X` is root), rotate `X` about its parent. This will make `X` be root. See Figure 12.1.

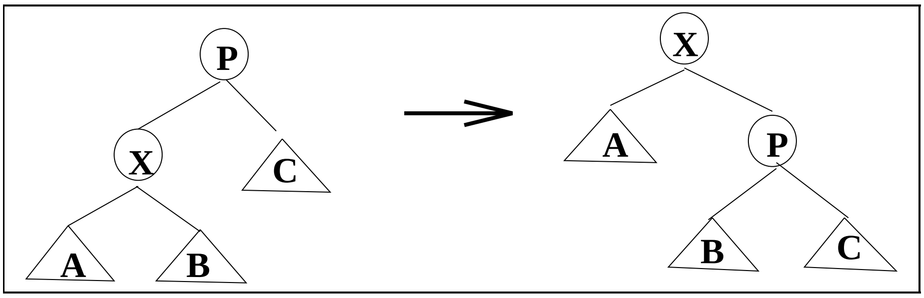

Figure 12.1: Node Has No Grandparent

3. If `X` has a grandparent:
   (a) If `X` and its parent are both left-children or both right-children, rotate the parent about the grandparent, then rotate `X` about its parent. See Figure 12.2.

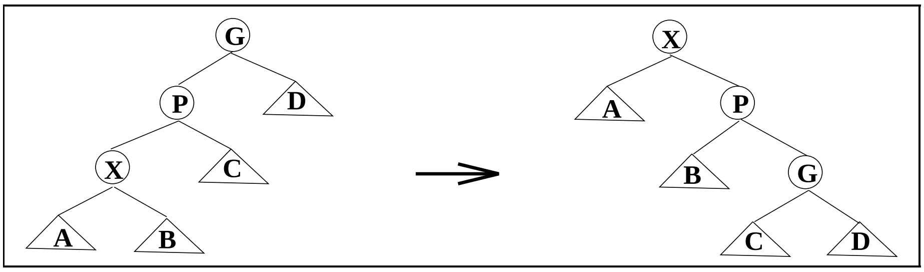

Figure 12.2: Node and Parent are Same Type

   (b) If `X` and its parent are opposite-type children (one is left, the other is right) rotate `X` about its parent, then rotate `X` about its new parent (*i.e.*, its former grandparent). See Figure 12.3.

12.4 Examples

Figure 12.4 shows the splay tree resulting from insertion of the Integers `1..4` in order. It then shows that splay tree after accessing the "`1`" node using the

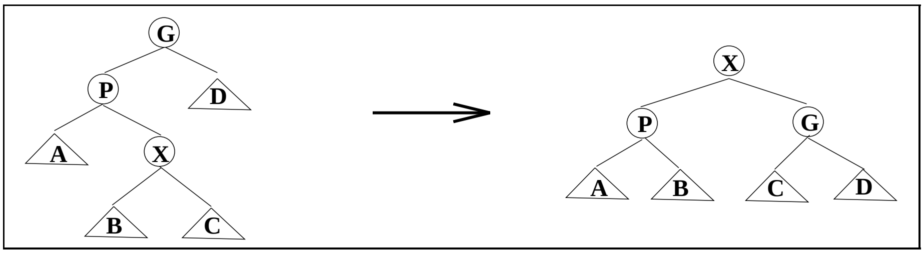

Figure 12.3: Node and Parent are Opposite Type

`Search` method. Note that the sequential insertions resulted in the same type of long tree that would have been obtained with ordinary binary search trees. In ordinary binary search trees, the sequence of four insertions would have taken $O(n)$ time per operation, dominated by the last insertion. This is not the case with the splay tree. Notice that each insertion was done just below the root because splaying moved the newly inserted node to the root. Thus, this particular sequence of four insertions was done in $O(1)$ per operation.

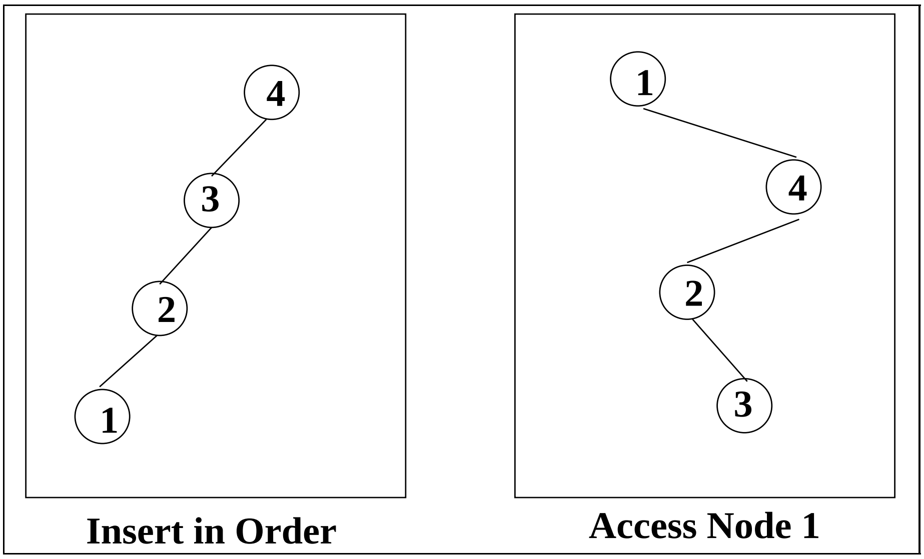

Figure 12.4: A Small Example

Figure 12.5 shows a splay tree before and after node ``X'' is accessed by an `Search` operation.

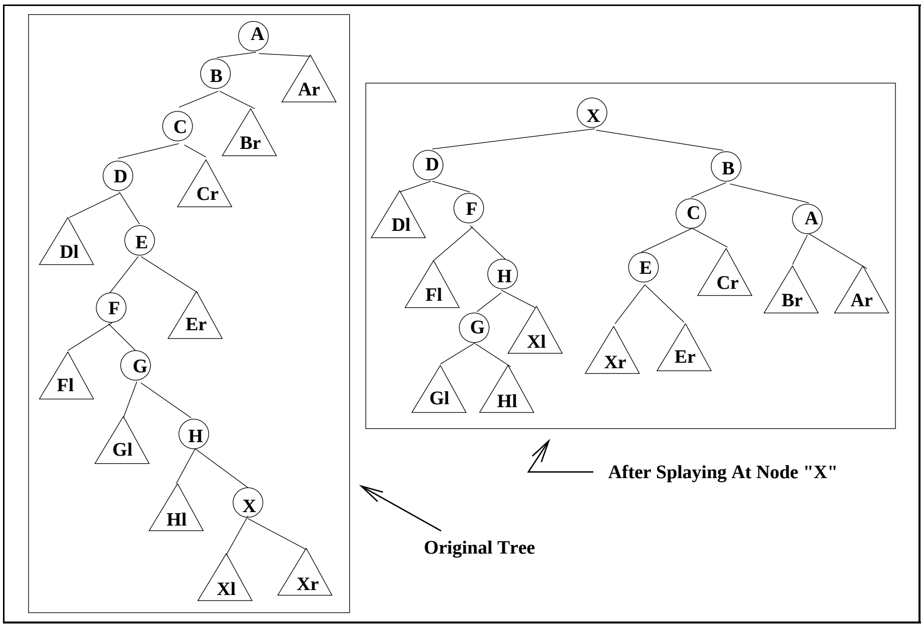

Figure 12.5: A Splaying Example

12.5 Implementation of Splay Trees

Here are some changes that might be made to the BST operations in the Heileman text for this course.

Splay trees are implemented as extensions of ordinary binary search trees. The efficiency of the splaying operation depends heavily on the ability to find the parent of a node. The `Bnode` class makes this an $O(1)$ operation by having each node store a reference to its parent.

Splaying takes place at a particular node, so a `Splay` method can be added as a member function to the `Bnode` class.

12.5.1 The `Search` Operation

The Heileman `BST::Search` method searches for a vertex with a given key, starting from a given vertex.

```
Bnode<T>* BST<T>::Search(Bnode<T>* u, int key, Boolean& success) const;
```

If the search succeeds, the vertex containing the key is returned. If the search fails, the last vertex visited is returned.

Here is the Heileman code for the other `BST::Search` method. It calls the overloaded `BST::Search` method described above.

```
1 template<class T>
2 T& BST<T>::Search(int key, Boolean& success) const
3
4   Bnode<T>* v = Search(root, key, success);
5   if (success == true)
6     return v->data;
7   else
8     return *(new T);  // unsuccessful search, return a dummy value
9
```

This method is easy to convert for use with splay trees. We simply splay the vertex `v` returned in line 4.

12.5.2 The `Predecessor` Operation

Here is Heileman's `Predecessor` operation (slightly modified) for ordinary binary search trees.

```
1   template<class T>
2   T& BST<T>::Predecessor(T & item, bool& success) const
3
4     Bnode<T>* v = Search(root, item, success);
5
6     if(success == false)
7       return *(new T);  // unsuccessful search, return dummy value
8
9     if (v->lchild != 0)
10      return SubtreeMax(v->lchild)->data;
11
12    Bnode<T>* original = v;
13    Bnode<T>* p = v->parent;
14    while (p != 0 && v == p->lchild)
15
16        v = p;
17        p = p->parent;
18
19
20    if (p == 0)
21      return *(new T); // no predecessor, return dummy value
22
23    return p->data;
24
```

To convert this for use with splay trees, we must perform splay operations as follows:

1. If the item does not exist in the tree, splay the vertex `v` returned by the search in line 4. There is no predecessor.

2. If the item does exist in the tree (the search succeeds):
   (a) If vertex `v` has a left child, splay the vertex returned by `SubTreeMax` on line 10.
   (b) Otherwise, we traverse the tree backwards, looking for the first vertex that does not have vertex `v` in its left subtree. Two cases can happen:
      i. There is no predecessor (line 21 is hit). In this case, splay the `original` vertex.
      ii. A predecessor is found. It will be `v` in the code. Splay it before returning at line 23.

12.5.3 The `Insert` Operation

Insertion in a splay tree is done in precisely the same way as in an ordinary binary search tree. This insertion method does not allow duplicate elements to be inserted. If the element to be inserted is already in the tree, then the insertion is not done, but the node holding the duplicate element is splayed. If the insertion succeeds, then the newly-inserted node is splayed.

Here is the Heileman code for the `Insert` method.

```
1  template<class T>
2  Boolean BST<T>::Insert(T item)
3
4    Boolean found;
5    Bnode<T>* new_node = new Bnode<T>(item);
6    assert(new_node != 0);
7    if (root == 0)
8      root = new_node;
9    else
10    Bnode<T>* v = Search(root, item.Key(), found);
11    if (found == true)   // item is already in tree
12        delete new_node;
13        return false;
14
15    if (item.Key() < v->data.Key())
16      BinaryTree<T>::InsertLeaf(new_node, v, left);
17    else
18      BinaryTree<T>::InsertLeaf(new_node, v, right);
19
20  return true;
21
```

Notice that it first searches for the item, starting at the root. If the search succeeds, the tree already contains the item and no insertion occurs. To use this method for use with splay trees, we must splay the vertex `v` returned in line 10.

If the tree does not already contain the item, we proceed with the insert operation from line 15. It's still vertex `v` that will be splayed and we do so before returning on line 20.

12.5.4 The `Delete` Operation

The `Delete` operation for a splay tree differs markedly from that in an ordinary BST. In fact, it's a lot simpler. If the item to be deleted is not in the tree, the vertex closest to it is splayed and we are done. On the other hand, if the item is in the tree, its vertex is splayed and then deletion occurs as described below. In either event, a `Search` for the item will result in the appropriate vertex being splayed to the root.

Let's look at the case in which the item is in the tree. The splay that resulted from the search operation has put that item's vertex at the root. In general, the vertex has both left and right children. Do the following operations:

1. Disconnect the left and right subtrees from the root. Call these T_L and T_R, respectively. Discard the root vertex.
2. Splay the maximum item in T_L. This will result in the root of T_L having no right child. (The largest element is at the root, so the root has no larger elements - no right subtree.)
3. Splay the minimum item in T_R. This will result in the root of T_R having no left child. (The smallest element is at the root, so the root has no smaller elements - no left subtree.)
4. All the elements in T_L are smaller than the elements in T_R. Make T_R be the right subtree of the root of T_L (or vice-versa).

Notice that there are three splays done.

1. The splay done when searching for the item in the tree.
2. The splay done in the left subtree T_L.
3. The splay done in the right subtree T_R.

There are no "cases" to worry about as in the ordinary BST deletion method.

12.6 Performance Considerations

The `Insert` operation uses ordinary `BST` insertion, then splays the inserted node to the root. Clearly the insertion portion of this operation has the same asymptotic behavior as that for `BST`s, namely $O(d)$. What about the splaying portion? If the work required at each step of the splay is independent of the number of nodes in the tree, the splay portion will be in $O(1)$ per step. The number of steps is bounded by the depth of the tree, so the splaying portion would also be in $O(d)$.

Well, is the splaying portion independent of the number of nodes in the tree? Look at what's involved:

1. Find the parent of the node and perhaps its parent too.
2. Determine if the node and its parent are the same type (left or right).
3. Rearrange the local structure of the tree.

Determining the type and rearranging the structure are local operations involving only a few nodes (the node, its parent, its grandparent). These operations must be in $O(1)$.

The cost of determining the parent of a node depends on how the parent is found. Since each node stores a reference to its parent, the cost is in $O(1)$, keeping the splay operation in $O(d)$.

12.6.1 Amortized Cost

For the average `BST`, $d = O(\lg n)$, so on average, operations on `BST`s are in $O(\lg n)$. A sequence of m operations (`Insert`, `Delete`, and `Search`) will cost $O(m \lg n)$.

The situation with splay trees is different. The structure of the tree is changed with each operation, so the notion of average is not relevant. A technique known as "amortized analysis" is used to show that over a sequence of m operations (starting with an empty tree) the cost of the sequence will be $O(m \lg n)$. This is the same cost as that for `BST`s so overall performance is the same. What's different is that the splay tree does not guarantee $O(\lg n)$ performance for *each* operation.