Notes on Splay Trees

D. Sleator Oct 9, 1998

These notes just describe the bottom-up splaying algorithm, the proof of the access lemma, and a few applications. (See www.cs.cmu.edu/~sleator for code and a demo of top-down splaying.)

Every time a node is accessed in a splay tree, it is moved to the root of the tree. The amortized cost of the operation is O(log n). Just moving the element to the root by rotating it up the tree does not have this property. The movement is done in a very special way that guarantees this amortized bound.

I'll describe the algorithm by giving three rewrite rules in the form of pictures. In these pictures, x is the node that was accessed (that will eventually be at the root of the tree). By looking at the local structure of the tree defined by x, x's parent, and x's grandparent we decide which of the following three rules to follow. We continue to apply the rules until x is at the root of the tree:

```
                             y             x
   zig (terminal case):     /     ====>     \
                           x                 y

                    z              z
                   /              /             x
   zig-zag:       y     ====>    x   ====>     / \
                   \            /             y   z
                    x          y

                    z                         x
                   /            y              \
   zig-zig:       y     ====>  / \   ====>      y
                 /            x   z              \
                x                                 z
```

Notes (1) Each rule has a mirror image variant, which covers all the cases.

(2) The zig-zig rule is the one that distinguishes splaying from just rotating x to the root of the tree.

(3) Top-down splaying is much more efficient in practice. You can find code for this at www.cs.cmu.edu/~sleator

Here are some examples:

```
            6                0                        3
           /                  \                      / \
          5                    5                    /   \
         /                    / \                  /     \
        4     splay(0)       3   6    splay(3)    0       5
       /      =======>      / \       =======>     \     / \
      3                    1   4                    1   4   6
     /                      \                        \
    2                        2                        2

   /
  1
 /
0
```


To analyze the performance of splaying, we start by assuming that each node x has a weight w(x) > 0. These weights can be chosen arbitrarily. For each assignment of weights we will be able to derive a bound on the cost of a sequence of accesses. We can choose the assignment that gives the best bound. By giving the frequently accessed elements a high weight, we will be able to get tighter bounds on the running time. Note that the weights are only used in the analysis, and do not change the algorithm at all.

Before we can state our performance lemma, we need to define two more quantities. The size of a node x (denoted s(x)) is the total weight of all the nodes in the subtree rooted at x. The rank of a node x (denoted r(x)) is the log_2 of the size of x.

```
          s(x) = Sum (over y in the subtree rooted at x) of w(y)

          r(x) = log(s(x))
```

The potential function will just be the sums of the ranks of all the nodes in the tree.

Notes about this potential:

(1) Doing a rotation between a pair of nodes x and y only effects the ranks of the nodes x and y, and no other nodes in the tree.

(2) Assuming all the weights are 1, the potential of a balanced tree is O(n), and the potential of a long chain (most unbalanced tree) is O(n log n).

Access lemma: The amortized cost (number of rotations) of splaying a node x in a tree with root t is at most 3(r(t)-r(x))+1.

Proof:

First we need the following claim. Consider a three nodes in the following configuration located anywhere in a tree:

```
                              b
                             / \
                            a   c
```

Let a, b, c denote the sizes of these nodes, and r(a), r(b), and r(c) denote the ranks of these nodes.

Claim: r(a)+r(c)-2r(b)+2 <= 0

Proof: The above follows from the following (obtained by taking 2 to the power of each side):

$$\frac{4\,ac}{b^2} \quad \le \quad 1$$

But b=a+c+w(b), so b>a+c. So the above inequality follows from:

$$\frac{4ac}{(a+c)^2} \quad \le \quad 1$$

This follows from

$$4ac \;\le\; a^2 + 2ac + c^2$$

This follows from

$$0 \;\le\; a^2 - 2ac + c^2$$

This follows from

$$0 \;\le\; (a-c)^2$$

This follows from the fact that the square of any number is positive.
This completes the proof of the claim.

Now we can go back to proving the lemma. The approach we take is to show that the amortized cost the zig-zag and zig-zig steps are bounded by 3(r'(x) - r(x)). And that the amortized cost of the zig step is bounded by 3(r'(x) - r(x))+1. Here r'() represents the rank function after the step, and r() represents the rank function before the step.

When we sum these costs to compute the amortized cost for the entire splay operation, they telescope to:

3(r(t) - r(x)) +1.

Note that the +1 comes from the zig step, which can happen only once.

It remains to show these bounds for the individual steps. There are three cases. We let "ac" denote amortized cost of the step.

zig: ac = 1 + r'(x) + r'(y) - r(x) - r(y).

note that r'(x) = r(y), so these cancel.

ac = 1 + r'(y) - r(x)

but r'(y) < r'(x) because y is below x in the tree after the step. So:

ac < 1 + r'(x) - r(x)

Now, since r'(x) > r(x), we can add 2(r'(x)-r(x)) to the right side, and get:

ac < 1 + 3(r'(x) - r(x))

zig-zag: ac = 2 + r'(x) + r'(y) + r'(z) - r(x) - r(y) - r(z)

again we note r'(x) = r(z), so they cancel.

ac = 2 + r'(y) + r'(z) - r(x) - r(y)

Now we use the fact that r(x) < r(y), giving:

ac < 2 + r'(y) + r'(z) - 2r(x)

Rearrange terms, adding and subtracting 2r'(x) gives:

ac < 2(r'(x)-r(x)) + (r'(y)+r'(z)-2r'(x) + 2)

Now if we consider the tree at the end of the zig-zag step, and we apply the claim we observed above about that tree, we see that the term in parentheses is <= 0. Therefore we conclude that:

ac < 2(r'(x)-r(x))

Again, we artifically add r'(x)-r(x) to this to give:

ac < 3(r'(x)-r(x))

Zig-zig: ac = 2 + r'(x) + r'(y) + r'(z) - r(x) - r(y) - r(z)

again we note r'(x) = r(z), so they cancel.

ac = 2 + r'(y) + r'(z) - r(x) - r(y)

apply the fact that r'(x) > r'(y) and r(y) > r(x)

ac < 2 + r'(x) + r'(z) - 2r(x)

Adding and subtracting 2r'(x), and regrouping gives:

ac < 3(r'(x) - r(x)) + (r(x)+r'(z)-2r'(x)+2)

The last term in parentheses is <= 0 by applying our claim. We simply apply it to the intermediate tree (after one rotation). This gives the desired result:

ac < 3(r'(x) - r(x))

This completes the proof of the access lemma.

Balance Theorem: A sequence of m splays in a tree of n nodes takes time O(m log(n) + n log(n)).

Proof: We apply the access lemma with all the weights equal to 1. For a given splay, r(t) <= log(n), and r(x) >= 0. So the amortized cost of the splay is at most:

3 log(n) + 1

The bound the cost of the sequence we add this amount for each splay, then add the initial minus the final potential. The initial potential is at most n log(n), and the final potential is at least 0. This gives a bound of:

m (3 log(n) + 1) + n log(n) = O(m log(n) + n log(n))

Q.E.D.

We can prove more interesting results by using non-uniform weights.

Consider a static search tree T with n nodes. The depth of a node x is defined to be the number of nodes on the path from x to the root. This is also the number of comparisons done when searching for x in the tree. The cost measure described in the access lemma is the number of rotations done. The number of comparisons is at most 1 more than this. So when counting comparisions we use 3(r(t)-r(x))+2.

The following theorem shows that splay trees perform within a constant factor of any static tree.

Static Optimality Theorem: Let T be any static search tree with n nodes. Let t be the number of comparisons done when searching for all the nodes in a sequence s of accesses. (This sum of the depths of all the nodes). Assuming all nodes of T are accessed at least once in s, the cost of splaying sequence s (starting with any initial splay tree) is O(t).

Proof: Envision an infinite binary tree. Assign a weight of 3^(-d) to a node depth d. It's easy to see that the weight of the whole tree is 1. Now we prune this tree (by removing branches) so that it looks like the tree T. This only removes weight, so the total weight is < 1. Now we can use this weight assignment to get a bound on the number of comparisons done by the splay algorithm.

If we splay a node of weight 3^(-d) (which is of depth d in T), then the amortized number of comparisons (by the access lemma) is at most

```
                 3 (log(1) - log (3^(-d))) + 2

              =  (3 log(3)) d + 2
```

Now, what's maximum decrease in potential? The initial potential of the tree is <= 0 because the size of all nodes is at most 1.

```
      log 3^(-d) <= rank of a node of depth d in T
```

or

```
     - d log(3)  <= rank of a node of depth d in T
```

Summing this, we get:

```
     Sum   -d(x) log(3)  <=  potential
    x in T
```

where d(x) is the depth of node x in T.
Applying this to the final potential we get:

```
  initial potenatial - final potential <=  log(3) *  Sum d(x)
                                                    x in T
```

We can put all this together to get a bound on the cost of splaying the whole sequence s. Let d(s(i)) be the depth of s(i), the ith element in the sequence s. Since each element is accessed at least once we can write the potential difference above as follows:

```
  initial potential - final potential <= log(3) * Sum d(s(i))
                                                   i
```

Let C(s) be the number of comparisons done by the splaying the sequence.

```
  C(s) <= initial potential - final potential + 3log(3) Sum (d(s(i)) + 2)
                                                         i
```

Since all the depths are at least 1, we can write this as:

```
  C(s) = O(Sum d(s(i)))  =  O(t)
             i
```


Q.E.D.

Note that to get this bound it's necessary to assume that all the elements are accessed at least once. Otherwise the splay tree could be initially a tree with millions of nodes near the top that are never accessed.