Self-Adjusting Search Trees

Jonathan Turner

April 16, 2013

We've seen how we can use explicit balance conditions to limit the height of binary search trees to at most $2 \lg n$. This leads to efficient operations on search trees, and speeds up algorithms that use them. In this section we will study a class of search trees that uses no explicit balance condition allowing it to produce search trees that are very unbalanced. Still, we can execute any sequence of m operations on these trees in $O(m \log n)$ time, making them competitive with search trees that are strictly balanced. Moreover, in some contexts they actually out-peform search trees that do have an explicit balance condition.

The key to the self-adjusting search tree is the *splay* operation, which restructures the tree in a way that makes later operations more efficient. The operation *splay*(x) is implemented by repeating the following step until x is the root of the tree.

> *Splay step.* If x has a grandparent, do the double rotation *rotate*$^2(x)$. Otherwise, do a single rotation *rotate*(x).

Note that the single rotation is only used on the last step. This is illustrated in Figure 1.

So, why is this a useful thing to do? First, notice that each double rotation reduces *depth*(x) by two. Moreover, each of these steps also reduces the depth of the *descendants* of x by one or two. Consider for example when x is an outer grandchild. Note that subtree A moves two steps closer to the root, while subtree B moves one step closer. When x is an inner grandchild, both of its subtrees move one step closer. Consequently, for all descendants of x the splay operation reduces their distance to the root by roughly half the original depth of x, or more.

So clearly, the descendants of x benefit from the splay operation. What about the other vertices? Consider a node u that is on the path from x to the root before the operation. The first few splay steps may have no effect

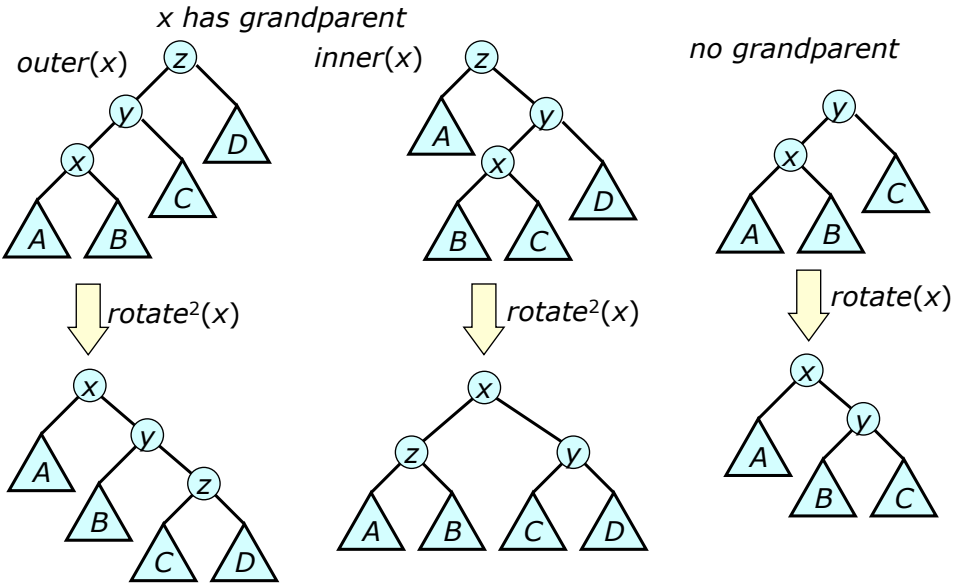

Figure 1: Splay steps

on u, but eventually a splay step occurs that moves x up past u, potentially moving u down by as much as two steps. But once this splay step occurs, u becomes a descendant of x and the remaining double rotations just move u closer to the root. The net reduction is in this case is nearly half the original depth of u.

If v is node that is not on the path from x to the root, let u be the nearest common ancestor of u and x. Here again, the first few double-rotations have no effect on v, and the step that moves x past u may increase the depth of v by 2. The remaining double rotations reduce the depth of v. The net reduction is roughly half the original depth of u. So, while some nodes will experience a small increase in their distance from the root, others will experience a large decrease.

A *splay* is performed as part of any operation that traverses a path from the root to a vertex x. The rationale is that the time required for the *splay* increases the cost of the original operation by no more than a constant factor, so we might as well restucture the tree so as to speed up later operations. This is illustrated below for the *access* operation.

item function *access*(**key** k, **set** s)
  do $key(s) < k$ **and** $left(s) \neq$ **null** $\Rightarrow$ $s := left(s)$
  | $k > key(s)$ **and** $right(s) \neq$ **null** $\Rightarrow$ $s := right(s)$

```
        od;
        s := splay(s);
        if key(s) = k  ⇒  return s
        | key(s) ≠ k  ⇒  return null
        fi;
    end;
```

The *splay* operation returns the new root of the tree. Note that the *splay* is performed, regardless of whether the desired key value is present in the tree.

The *insert* operation works just like in the case of unbalanced trees. The only difference is the *splay* preceding the return.

```
    function insert(item i, sorted set s)
        item x; x := s;
        if s = null  ⇒  return i fi;
        do key(i) < key(x) and left(x) ≠ null  ⇒  x := left(x)
        |  key(i) > key(x) and right(x) ≠ null  ⇒  x := right(x)
        od;
        if key(i) = key(x) ⇒ splay(x) return null
        | key(i) < key(x) ⇒ left(x) := i;
        | key(i) > key(x) ⇒ right(x) := i;
        fi;
        p(i) := x  left(i), right(i) := null;
        return splay(i);
    end;
```

The *delete* operation is also extended to include a *splay*. If the delete is done at a node at a node x with one or two children, the *splay* is done at the original parent of x. If x has two children, the *splay* is done at the original parent of the node that takes x's place in the tree.

The *join* does not require a *splay*. The *split* operation starts with a splay, that brings the split node to the root of the tree. At that point, it's trivial to divide the tree into the required three components.

All operations that take more than constant time include a *splay* and their running time is bounded by the number of splay steps. Hence, we can determine the running time of a sequence of operations by counting the total number of splay steps.