15-451 Algorithms
D. Sleator

Splay Trees

-- binary search trees in general
-- definition of splay trees
-- analysis of splay trees

Binary Search Trees

You have probably seen binary search trees before. You should understand what they are. Binary search trees is a class of data structures where

(1) Each node stores a piece of data
(2) Each node has two pointers to two other binary search trees
(3) The overall structure of the pointers is a tree
    (there's a root, it's acyclic, and every node is
     reachable from the root.)

Binary search trees are a way to store a update a set of items, where there is an ordering on the items. I know this is rather vague. But there is not a precise way to define the gamut of applicatons of search trees. In general, there are two classes of applications. Those where each item has a key value from a totally ordered universe, and those where the tree is used as an efficient way to represent an ordered list of items.

Some applications of binary search trees:

Storing a set of names, and being able to lookup based on a prefix of the name. (Used in internet routers.)

Storing a path in a graph, and being able to reverse any subsection of the path in O(log n) time. (Useful in travelling salesman problems).

Being able to quickly determine the rank of a given node in a set.

You should know what a rotation in a tree is.

```
              1                                  2
             / \      left rotation at 1        / \
            0   2     ------------------->     1   4
                 \                            /   / \
                  4                          0   3   5
                 / \
                3   5
```

Splay Trees (self-adjusting search trees)

These notes just describe the bottom-up splaying algorithm, the proof of the access lemma, and a few applications.

(For more information and a splay tree demo is see http://www.link.cs.cmu.edu/splay/)

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. Splay trees do 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. Code for
          doing this is on my web site (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.

(A commonly used case is to assign all the weights to be 1.)

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 floor(log_2) of the size of x. Restating these:

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

r(x) = floor(log(s(x)))

For each node x, we'll keep r(x) tokens on that node. (Alternatively,
the potential function will just be the sums of the ranks of all the
nodes in the tree.)

Here's an example to illustrate this: Here's a tree, labeled with sizes
on the left and ranks on the right.

```
               9 o                           3 o
                /                             /
             8 o                           3 o
              / \                           / \
           6 o   o 1                     2 o   o 0
            / \                           / \
         2 o   o 3                     1 o   o 1
          /   /                         /   /
       1 o   o 2                     0 o   o 1
              \                             \
               o 1                           o 0
```

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.
Furthermore, if y was x's parent before the rotation, then the
rank of y before the rotation equals the rank of x after the
rotation.

(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).

(3) In the banker's view of amortized analysis, we can
think of having r(x) tokens on node x.

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

Proof:

As we do the work, we must pay one token for each splay step we do.
Furthermore we must make sure that there are always r(x) tokens on node
x as we do our restructuring. We are going to allocate 3(r(t) - r(x)) +1
tokens to do the splay. Our job in the proof is to show that this is
enough.

```
First we need the following observation about ranks, called the Rank
Rule.  Suppose that two siblings have the same rank, r.  Then the parent
has rank at least r+1.  This is because if the rank is r, then the size
is at least 2^r.  If both siblings have size at least 2^r, then the
total size is at least 2^(r+1) and we conclude that the rank is at least
r+1.  We can represent this with the following diagram:

                 x
                / \       Then x >= r+1
               r   r

Conversly, suppose we find a situation where a node and its parent have
the same rank, r.  Then the other sibling of the node must have rank < r.
So if we have three nodes configured as follows, with these ranks:

                 r
                / \       Then x < r
               x   r

Now we can go back to proving the lemma.  The approach we take is to
show that 3(r'(x) - r(x)) tokens are sufficient to pay for the
zig-zag or a zig-zig steps.  And that 3(r'(x) - r(x)) +1 is sufficient
to pay for the zig step.  (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.

Here's an example, the labels are ranks:

           2 o                     2 x
            /                       / \
         2 o                     0 o   o 2
          /                             /
       2 o       splay(x)          2 o
        /       ==========>         / \
     2 x                         1 o   o 0
      / \                         /
   0 o   o 1                   0 o
        /
       o 0

-----------------            ----------------
Total:  9                          7

We allocated:                   3(2-2)+1 = 1
extra tokens from restructuring:   9-7   = 2
                                       ------
                                            3

There are 2 splay steps.  So 3 > 2, and we have enough.

It remains to show these bounds for the individual steps.  There are
three cases, one for each of the types of splay steps.

Zig:
               r o              o r
                /       ==>      \
             a o                    o b <= r
```

The actual cost is 1, so we spend one token on this.
We take the tokens on a and augment them with another
r-a and put them on b. Thus the total number of tokens
needed is 1+r-a. This is at most 1+3(r-a).

Zig-zag: We'll split it into 2 cases:

Case 1: The rank does not increase between the starting
node and ending node of the step.

```
                r o                    o r
                 /                    / \
              r o         ===>     a o   o b
                 \
                r o
```

By the Rank Rule, one of a or b must be < r, so
one token is released from the data structure.
we use this to pay for the work.

Thus, our allocation of 3(r-r) = 0 is sufficient
to pay for this.

Case2: (The rank does increase)

```
                r o                  o r
                 /                  / \
              b o         ===>   c o   o d
                 \
                a o
```

The tokens on c can be supplied from those on b.
(There are enough there cause b >= c.)

Note that r-a > 0. So:

use r-a (which is at least 1) to pay for the work
use r-a to augment the tokens on a to make enough for d

Summing these two gives: 2(r-a). But 2(r-a) <= 3(r-a), which
completes the zig-zag case.

Zig-zig: Again, we split into two cases.

Case 1: The rank does not increase between the starting node and the
ending node of the step.

```
         r o                r o                 o r
          /                  / \                 \
       r o      =====>    r o   o d   ====>       o c <= r
        /                                          \
     r o              (d<r by rank rule)            o d < r
```

As in the zig-zag case, we use the token gained because
d < r to pay for the step. Thus we have 3(r-r)=0 tokens
and we need 0.

Case 2: The rank increases during the step.

```
          r o                                 o r
           /                                   \
        b o          ============>              o c <= r
         /                                       \
      a o                                         o d <= r
```

```
  use r-a (which is at least 1) to pay for the work
  use r-a to boost the tokens on a to cover those needed for d
  use r-a to boost the tokens on b to cover those needed for c

  Summing these gives 3(r-a), which completes the analysis
  of the zig-zig case.
```

This completes the proof of the access lemma. QED

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:

$$3 \log(n) + 1$$

We now switching to the world of potentials (potential = total tokens in
the tree). To 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). The cost of splaying that sequence of requests, starting
with any initial splay tree is O(n^2 + t).

Dictionary Implementation with Splay Trees
(Notes courtesy of Randy Bryant)

Standard operations we want to perform:

```
Insert(T, x): Insert key x into tree T
Delete(T, x): Delete key x from tree T
Lookup(T, x): Retrieve data associated with key x from tree.

Assume we implement operation Splay(T,x) that rearranges the nodes in
T to have a root value y, where:

  If x is in T, then y = x.
  If x is not in T, then either:
     y is the largest element > x in T, OR
     y is the smallest element < x in T.

This is exactly what the implementation of top-down splaying does.
You can find this here: www.link.cs.cmu.edu/splay/, along with
a demo of top-down splaying.

Want to show that any sequence of n dictionary operations has cost O(n
log n).  We the analysis where the s(x) equals the number of nodes in
the subtree rooted by x.  Of course, the tree will never have more than n
nodes.  We will make sure that the operations are performed with a
fixed number of splay operations, plus other steps having constant
time.  We will may need to add additional weight to the tree as well.

Implementation:

Insert(T, x):
  Perform Splay(T,x).  If x is at root, then already in tree.
  Otherwise have one of two cases:

       y < x              x
      / \        -->     / \
     A   B              y   B
                       /
                      A

       y > x              x
      / \        -->     / \
     A   B              A   y
                             \
                              B

  Cost: 1 splay operation, plus addition of node x, having r(x) <= log n.

Delete(T,x):

  Perform Splay(T,x).  If x is not at root, then no need to delete.  Otherwise
  have two subtrees A and B.

     x
    / \
   A   B

  If A is empty, then just return B.
  Otherwise, perform Splay(A,infinity).  This creates tree:

     y
    /
   C

  Where y is the largest element in T that is < y.  Final result is:
     y
    / \
   C   B
```

Cost: 2 splay operations. Have r(C) <= r(A), so potential for tree does not increase.

Lookup(T,x):

Perform Splay(T,x). Examine root.

Cost: 1 splay operation.

So, sequence of n dictionary operations involves:

```
  <= 2n splay operations on trees of size <= n:             O(n log n) cost.
  <= cn cost to examine/modify/add root:                    O(n) cost.
  <= n additions of new nodes adding <= log n to potential: O(n log n) cost.
```