CS61B: Lecture 36
Wednesday, April 23, 2014

Today's reading: Goodrich & Tamassia, Section 10.3.

SPLAY TREES

A splay tree is a type of balanced binary search tree. Structurally, it is identical to an ordinary binary search tree; the only difference is in the algorithms for finding, inserting, and deleting entries.

All splay tree operations run in O(log n) time _on_average_, where n is the number of entries in the tree. Any single operation can take Theta(n) time in the worst case. But any sequence of k splay tree operations, with the tree initially empty and never exceeding n items, takes O(k log n) worst-case time.

Although 2-3-4 trees make a stronger guarantee (_every_ operation on a 2-3-4 tree takes O(log n) time), splay trees have several advantages. Splay trees are simpler and easier to program. Because of their simplicity, splay tree insertions and deletions are typically faster in practice (sometimes by a constant factor, sometimes asymptotically). Find operations can be faster or slower, depending on circumstances.

Splay trees are designed to give especially fast access to entries that have been accessed recently, so they really excel in applications where a small fraction of the entries are the targets of most of the find operations.

Splay trees have become the most widely used basic data structure invented in the last 30 years, because they're the fastest type of balanced search tree for many applications.

Tree Rotations

```
Like many types of balanced search         Y                             X
trees, splay trees are kept balanced      / \        rotate left        / \
with the help of structural changes      X   ^      <------------      ^   Y
called _rotations_.  There are two      / \ /C\                       /A\ / \
types--a left rotation and a right     ^  ^         ------------>         ^  ^
rotation--and each is the other's     /A\/B\         rotate right        /B\/C\
reverse.  Suppose that X and Y are
```

binary tree nodes, and A, B, and C are subtrees. A rotation transforms either of the configurations illustrated above to the other. Observe that the binary search tree invariant is preserved: keys in A are less than or equal to X; keys in C are greater than or equal to Y; and keys in B are >= X and <= Y.

Rotations are also used in AVL trees and red-black trees, which are discussed by Goodrich and Tamassia, but are not covered in this course.

Unlike 2-3-4 trees, splay trees are not kept perfectly balanced, but they tend to stay reasonably well-balanced most of the time, thereby averaging O(log n) time per operation in the worst case (and sometimes achieving O(1) average running time in special cases).

Splay Tree Operations

[1] Entry find(Object k);

The find() operation in a splay tree begins just like the find() operation in an ordinary binary search tree: we walk down the tree until we find the entry with key k, or reach a dead end (a node from which the next logical step leads to a null pointer).

However, a splay tree isn't finished its job. Let X be the node where the search ended, whether it contains the key k or not. We _splay_ X up the tree through a sequence of rotations, so that X becomes the root of the tree. Why? One reason is so that recently accessed entries are near the root of the tree, and if we access the same few entries repeatedly, accesses will be very fast. Another reason is because if X lies deeply down an unbalanced branch of the tree, the splay operation will improve the balance along that branch.

When we splay a node to the root of the tree, there are three cases that determine the rotations we use.

```
-1-  X is the right child of a left      G               G               X
child (or the left child of a right     / \             / \             / \
child):  let P be the parent of X,     P   ^           X   ^           P   G
and let G be the grandparent of X.    / \ /D\  ==>    / \ /D\  ==>    / \ / \
We first rotate X and P left,        ^  X            P  ^            ^  ^ ^  ^
and then rotate X and G right, as   /A\/ \          / \/C\          /A\/BVC\/D\
illustrated at right.                  ^  ^        ^  ^
                                      /B\/C\      /A\/B\     Zig-Zag
```

The mirror image of this case--where X is a left child and P is a right child--uses the same rotations in mirror image: rotate X and P right, then X and G left. Both the case illustrated above and its mirror image are called the "zig-zag" case.

```
-2-  X is the left child of a left     G                P               X
child (or the right child of a right  / \              / \             / \
child):  the ORDER of the rotations  P   ^            X   G           ^   P
is REVERSED from case 1.  We        / \ /D\  ==>    / \ / \    ==>  /A\ / \
start with the grandparent,        X  ^            ^  ^ ^  ^           ^  G
and rotate G and P right.         / \/C\          /A\/BVC\/D\         /B\/ \
Then, we rotate P and X right.   ^  ^                                    ^  ^
                                /A\/B\                     Zig-Zig     /C\/D\
```

The mirror image of this case--where X and P are both right children--uses the same rotations in mirror image: rotate G and P left, then P and X left. Both the case illustrated above and its mirror image are called the "zig-zig" case.

We repeatedly apply zig-zag and zig-zig rotations to X; each pair of rotations raises X two levels higher in the tree. Eventually, either X will reach the root (and we're done), or X will become the child of

```
the root.  One more case handles the latter                P             X
circumstance.                                              / \           / \
                                                          X   ^         ^   P
-3-  X's parent P is the root:  we rotate X and P        / \ /C\  ==>  /A\ / \
so that X becomes the root.  This is called the         ^  ^              ^  ^
"zig" case.                                            /A\/B\     Zig     /B\/C\
```

Here's an example of "find(7)". Note how the tree's balance improves.

```
    11                     11                       11                  [7]
   /  \                   /  \                     /  \                 / \
  1    12                1    12                 [7]   12              1   11
 / \                    / \                      / \                   /\   / \
0   9                  0   9                    1   9                 0 5   9  12
   / \                    / \                  / \ / \                 / \ / \
  3   10  =zig-zig=>    [7]  10  =zig-zag=>  0  5 8  10   =zig=>     3  6 8  10
 / \                    / \                    / \                   / \
2   5                  5   8                  3   6                 2   4
   / \                / \                    / \
  4  [7]             3   6                  2   4
     / \            / \
    6   8          2   4
```

```
By inspecting each of the three cases (zig-zig, zig-zag, and zig), you can
observe a few interesting facts.  First, in none of these three cases does the
depth of a subtree increase by more than
two.  Second, every time X takes two                      9
steps toward the root (zig-zig or zig-zag),              / \
every node in the subtree rooted at X moves             8   10
at least one step closer to the root.                  /
As more and more nodes enter X's subtree,             7
more of them get pulled closer to the root.          /
                                                    6           1
A node that initially lies at depth d on           /           / \
the access path from the root to X moves          5           0   8
to a final depth no greater than 3 + d/2.        /               / \
In other words, all the nodes deep              4               6   9
down the search path have their                /               / \   \
depths roughly halved.  This tendency         3  ==========>  4   7   10
of nodes on the access path to move          /     find(1)   / \
toward the root prevents a splay tree       2               2   5
from staying unbalanced for long           /                 \
(as the example at right illustrates).    1                   3
                                         /
[2]  Entry min();                       0
     Entry max();
```

These methods begin by finding the entry with minimum or maximum key, just like in an ordinary binary search tree. Then, they splay the node containing the minimum or maximum key to the root.

```
[3]  Entry insert(Object k, Object v);
```

insert() begins by inserting the new entry (k, v), just like in an ordinary binary search tree. Then, it splays the new node to the root.

```
[4]  Entry remove(Object k);
```

An entry having key k is removed from the tree, just as with ordinary binary search trees. Recall that the node containing k is removed if it has zero or one children. If it has two children, the node with the next higher key is removed instead. In either case, let X be the node removed from the tree. After X is removed, splay X's parent to the root. Here's a sequence illustrating the operation remove(2).

```
                      2             4               5
                     / \           / \             / \
                    1   7         1   7           4   7
                       / \   ==>     / \   ==>   /     \
                      5   8         5   8       1       8
                     /
                    4
```

In this example, the key 4 moves up to replace the key 2 at the root. After the node containing 4 is removed, its parent (containing 5) splays to the root.

If the key k is not in the tree, splay the node where the search ended to the root, just like in a find() operation.

Postscript: Babble about Splay Trees (not examinable, but good for you)

It may improve your understanding to watch the splay tree animation at http://www.ibr.cs.tu-bs.de/courses/ss98/audii/applets/BST/SplayTree-Example.html .

Splay trees can be rigorously shown to run in O(log n) average time per operation, over any sequence of operations (assuming we start from an empty tree), where n is the largest size the tree grows to. However, the proof is quite elaborate. It relies on an interesting algorithm analysis technique called _amortized_analysis_, which uses a _potential_function_ to account for the time saved by operations that execute more quickly than expected. This "saved-up time" can later be spent on the rare operations that take longer than O(log n) time to execute. By proving that the potential function is never negative (that is, our "bank account" full of saved-up time never goes into the red), we prove that the operations take O(log n) time on average.

The proof is given in Goodrich & Tamassia Section 10.3.3 and in the brilliant original paper in the Journal of the Association for Computing Machinery, volume 32, number 3, pages 652-686, July 1985. Unfortunately, there's not much intuition for why the proof works. You crunch the equations and the result comes out.

In 2000, Danny Sleator and Robert Tarjan won the ACM Kanellakis Theory and Practice Award for their papers on splay trees and amortized analysis. Splay trees are used in Windows NT (in the virtual memory, networking, and file system code), the gcc compiler and GNU C++ library, the sed string editor, Fore Systems network routers, the most popular implementation of Unix malloc, Linux loadable kernel modules, and in much other software.

```
                                                                          .
                                                                         .
                                                                        .
When do operations occur that take more than O(log n) time?            /
Consider inserting a long sequence of numbers in order:  1, 2, 3,     4
etc.  The splay tree will become a long chain of left children (as   /
illustrated at right).  Now, find(1) will take Theta(n) time.       3
However, each of the n insert() operations before the find took O(1)   /
time, so the average for this example is O(1) time per operation.  2
                                                                   /
                                                                  1
```

The fastest implementations of splay trees don't use the bottom-up splaying strategy discussed here. Splay trees, like 2-3-4 trees, come in bottom-up and top-down versions. Instead of doing one pass down the tree and another pass up, top-down splay trees do just one pass down. This saves a constant factor in the running time.

There is an interesting conjecture about splay trees called the _dynamic_ _optimality_conjecture_: that splay trees are as asymptotically fast on _any_ sequence of operations as _any_ other type of search tree with rotations. What does this mean? Any sequence of splay tree operations takes amortized O(log n) time per operation, but sometimes there are sequences of operations that can be processed faster by a sufficiently smart data structure. One example is accessing the same ten keys over and over again (which a splay tree can do in amortized O(1) time per access). The dynamic optimality conjecture guesses that if _any_ search tree can exploit the structure of a sequence of accesses to achieve asymptotically faster running time, so can splay trees.

The conjecture has never been proven, but it's not clear whether it's been disproven, either.

One special case that has been proven is that if you perform the find operation on each key in a splay tree in order from the smallest key to the largest key, the total time for all n operations is O(n), and not O(n log n) as you might expect.