CS61B Summer 2006
Instructor: Erin Korber
Lectures 18,19: 26,27 July

Yesterday, we talked about binary search trees, and their efficiency over linked lists as a data structure for storing dictionaries. However, the possible $\Theta(n)$ worst-case performance for most operations (which occurs when the tree is highly unbalanced) was a concern. Today, we'll discuss several ways to augment the BST data structure to prevent this situation from occuring.

1 AVL Trees

AVL trees are the first example (invented in 1962) of a "self-balancing" binary search tree. AVL trees satisfy the *height balance property* - that for any node n, the heights of n's left and right subtrees can differ by at most 1. This property will ensure that the tree remains balanced, so its height will be proportional to log(n) as we want, avoiding the possible worst case where the height of the tree was proportional to n. Note that the height balance property enforces that any subtree of an AVL tree is also an AVL tree. So for an AVL tree, the running time of the `find, insert`, and `remove` operations will be in O(log n), even in the worst case (unlike a generic BST, in which the worst case time was in $\Theta(n)$).

We will need to modify our `insert` and `remove` methods to ensure that we maintain the height balance property when inserting and removing nodes. Our job will be made easier if we add a `height` instance variable to our nodes, so we can easily tell the height of a node and compare it with others. We will call a node *unbalanced* if its left and right children differ in height by more than 1.

We will balance the tree after an insertion with a search-and-repair strategy called a "trinode restructing" method, which will alter the tree in all the needed ways to maintain the height balance property (along with the BST invariant).

1.1 Trinode restructing method

After inserting a node w, follow the path up towards the root from it until you find an unbalanced node; call this node z. Call the higher-height child of z y, and call the higher-height child of y x. (So x is a grandchild of z.) Then execute the following:

1. Let (a,b,c) be an inorder listing of the nodes x, y, and z, and let (T0, T1, T2, T3) be an inorder listing of the four subtrees of x,y, and z that are not rooted at x, y, or z.
2. Replace the subtree rooted at z with a new subtree rooted at b.
3. Let a be the left child of b and let T0 and T1 be the left and right subtrees of a.
4. Let c be the right child of b and let T2 and T3 be the left and right subtrees of c.

Example: inserting the "54" node and rebalancing. Observe that the nodes for 78 and 44 become unbalanced after this insertion.

- x is the "62" node, y is the "50" node, and z is the "78" node.
- So a is the "50" node, b is the "62" node, and c is the "78" node.
- T0 is the subtree rooted at "48", T1 at "54", T2 is empty, and T3 at "88".

```
     44                              44
    /  \                            /  \
  17    78(z)                     17    62
 /     /   \                     /     /  \
12    50(y) 88                  12    50   78
     /  \                            /  \   \
    48   62(x)                      48  54   88
        /
       54
```

Often, the procedure done by the trinode restructing is presented as 4 different kinds of rotations, corresponding to the 4 possible ways that (a,b,c) could be mapped to (x,y,z).

Note that since the trinode restructuring method needs to look at and move only a fixed number of nodes, it runs in O(1) time, so the `insert` operation will still take only (log n) time.

Similarly to with insertion, we remove an item from an AVL tree by first removing in the usual way for a BST, then rebalancing the tree using trinode restructuring. However, this restructuring may reduce the height of the subtree rooted at b by 1, which could cause an ancestor of b to become unbalanced. Therefore, a single restructuring operation isn't sufficient. After rebalancing z, we continue walking up the tree towards to root, looking for unbalanced nodes and rebalancing every time we find one.

Although we have to do possibly more than a single restructuring, the most we could have to do is in O(log n), since the height of the tree is (log n). Therefore, `remove` is still in O(log n) time.

2 Splay trees

A splay tree is another type of balanced binary search tree. All splay tree operations run in O(log n) time *on average*, where n is the number of entries in the tree, assuming you start with an empty tree. Any single operation can take $\Theta(n)$ time in the worst case, but operations slower than O(log n) time happen rarely enough that they don't affect the average.

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

Splay trees have become the most widely used data structure invented in the last 20 years, because they're the fastest type of balanced search tree for many applications, since it is quite common to want to access a small number of entries very frequently, which is where splay trees excel.

Splay trees, like AVL trees, are kept balanced by means of rotations. Unlike AVL 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). We'll analyze this phenomenon more precisely when we discuss *amortized analysis.*

2.1 Splay tree operations

- `find(Key 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.

  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 quite 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 child (or the left child of a right child): let P be the parent of X, and let G be the grandparent of X. We first rotate X and P left, and then rotate X and G right, as illustrated below.

   ```
        G               G               X
       / \             / \             /  \
      P  /D\          X  /D\          P     G
     / \       ==>   / \    ==>      / \   / \
    /A\ X            P  /C\        /A\ /B|C\ /D\
       / \          / \
     /B\ /C\      /A\ /B\
   ```

   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 child (or the right child of a right child): the ORDER of the rotations is REVERSED from case 1. We start with the grandparent, and rotate G and P right. Then, we rotate P and X right.

   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 below and its mirror image are called the "zig-zig" case.

   ```
          G               P               X
         / \             / \             / \
        P  /D\          X   G          /A\  P
       / \      ==>    / \ / \    ==>      / \
      X  /C\          /A\/B|C\/D\        /B\  G
     / \                                       / \
   /A\ /B\                                   /C\ /D\
   ```

   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 circumstance.

3. X’s parent P is the root: we rotate X and P so that X becomes the root. This is called the ”zig” case.

```
          P             X
         / \           / \
        X  /C\       /A\  P
       / \      ==>      / \
     /A\ /B\           /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 steps toward the root (zig-zig or zig-zag), every node in the subtree rooted at X moves at least one step closer to the root. As more and more nodes enter X’s subtree, more of them get pulled closer to the root.

A node that initially lies at depth d on the access path from the root to X moves to a final depth no greater than 3 + d/2. In other words, all the nodes deep down the search path have their depths roughly halved. This tendency of nodes on the access path to move toward the root prevents a splay tree from staying unbalanced for long (as the example below illustrates).

```
              13
             /  \
            11   14         1
           / \             / \
          9   12          0   11
         / \                 /  \
        7   10              /    13
       / \                 7    /  \
      5   8  ==========>  / \ 12    14
     / \       find(1)   /   9
    3   6               3   / \
   / \                 / \ 8   10
  1   4               2   5
 / \                     / \
0   2                   4   6
```

- `first(), last()`

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

- `insert(KeyValPair p)`

  `insert` begins by inserting the new entry p, just like in an ordinary binary search tree. Then, it splays the new node to the root.

- `remove(Key k)`

  An entry having key k is removed from the tree, just as with ordinary binary search trees. Let X be the node removed from the tree. After X is removed, splay its 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 moved up to replace the key 2 at the root. After the node containing 4 was removed, its parent (containing 5) splayed 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.

3 2-3-4 Trees

2-3-4 trees are a kind of prefectly balanced search tree. They are so named because every node has 2, 3, or 4 children, except leaf nodes, which are all at the bottom level of the tree. Each node stores 1, 2, or 3 entries, which determine how other entries are distributed among its children's subtrees.

Each non-leaf node has one more child than keys. For example, a node with keys [20, 40, 50] has four children. Eack key k in the subtree rooted at the first child satisfies k $\leq$ 20; at the second child, 20 $\leq$ k $\leq$ 40; at the third child, 40 $\leq$ k $\leq$ 50; and at the fourth child, k $\geq$ 50.

3.1 B-trees: the general case of a 2-3-4 tree

2-3-4 trees are a type of B-tree. A B-tree is a generalized version of this kind of tree where the number of children that each node can have varies. Because a range of child nodes is permitted, B-trees do not need re-balancing as frequently as other self-balancing binary search trees, but may waste some space, since nodes are not entirely full. The lower and upper bounds on the number of child nodes are typically fixed for a particular implementation. For example, in a 2-3-4 tree, each non-leaf node may have only 2,3, or 4 child nodes. The number of elements in a node is one less than the number of children.

A B-tree is kept balanced by requiring that all leaf nodes are at the same depth. This depth will increase slowly as elements are added to the tree, but an increase in the overall depth is infrequent, and results in all leaf nodes being one more hop further removed from the root.

B-trees have advantages over alternative implementations when node access times far exceed access times within nodes. This usually occurs when most nodes are in secondary storage, such as on hard drives. By maximizing the number of child nodes within each internal node, the height of the tree decreases, balancing occurs less often, and efficiency increases. Usually this value is set such that each node takes up a full disk block or some other size convenient to the storage unit being used. So in practice, B-trees with larger internal node sizes are more commonly used, but we will be discussing 2-3-4 trees since it is useful to be able to work out examples with a managable node size.

3.2 2-3-4 tree operations

- `find(Key k)`

  Finding an entry is straightforward. Start at the root. At each node, check for the key k; if it's not present, move down to the appropriate child chosen by comparing k against the keys. Continue until k is found, or k is not found at a leaf node. For example, find(74) traverses the double-lined boxes in the diagram below.

```
               ==========
               +20 40 50+
            /--==========------\
       /---/      /  \          \-----\
       ----      ----      ----        =======
       |14|      |32|      |43|        +70 79+
       ----      ----      ----        =======
       /  \      /  \      /  \        /  |  \
---- ---- ---- ---- ---- ---- ---------- ==== ----
|10| |18| |25| |33| |42| |47| |57 62 66| +74+ |81|
---- ---- ---- ---- ---- ---- ---------- ==== ----
```

- `insert(KeyValPair p)`

  insert(), like find(), walks down the tree in search of the key k. If it finds an entry with key k, it proceeds to that entry's "left child" and continues.

  Unlike find(), insert() sometimes modifies nodes of the tree as it walks down. Specifically, whenever insert() encounters a 3-key node, the middle key is ejected, and is placed in the parent node instead. Since the parent was previously treated the same way, the parent has at most two keys, and always has room for a third. The other two keys in the 3-key node are split into two separate 1-key nodes, which are divided underneath the old middle key (as the figure illustrates).

```
        ----         -------
        |20|         |11 20|
        ----         -------
        /  \   =>    /  |  \
---------- ----    ---- ---- ----
|10 11 12| |30|    |10| |12| |30|
---------- ----    ---- ---- ----
```

For example, suppose we insert 60 into the tree depicted earlier. The first node traversed is the root, which has three children; so we kick the middle child (40) upstairs. Since the root node has no parent, a new node is created to hold 40 and becomes the root. Similarly, 62 is kicked upstairs when insert() finds the node containing it. This ensures us that when we arrive at the leaf node (labeled 57 in this case), there's room to add the new key 60.

```
                  ----
                  |40|
                /------\
           /---/        \----\
        ----                  ----
        |20|                  |50|
        ----                /------\
       /    \              /        \
   ----      ----      ----          ----------
   |14|      |32|      |43|          |62 70 79|
   ----      ----      ----          ----------
   /  \      /  \      /  \          /  |  |   \
---- ---- ---- ---- ---- ---- ------- ---- ---- ----
|10| |18| |25| |33| |42| |47| |57 60| |66| |74| |81|
---- ---- ---- ---- ---- ---- ------- ---- ---- ----
```

Observe that along the way, we created a new 3-key node "62 70 79". We do not kick its middle key upstairs until the next time it is traversed.

Again, the reasons why we split every 3-key node we encounter (and move its middle key up one level) are (1) to make sure there's room for the new key in the leaf node, and (2) to make sure that above the leaf nodes, there's room for any key that gets kicked upstairs. Sometimes, an insertion operation increases the depth of the tree by one by creating a new root.

- `remove(Key k)`

  2-3-4 tree remove() is similar to remove() on binary trees: you find the entry you want to remove (having key k). If it's in a leaf node, you remove it. If it's in an interior node, you replace it with the entry with the next higher key. That entry must be in a leaf node. In either case, you remove an entry from a leaf node in the end.

Like insert(), remove() changes nodes of the tree as it walks down. Whereas insert() eliminates 3-key nodes (moving nodes up the tree) to make room for new keys, remove() eliminates 1-key nodes (sometimes pulling keys down the tree) so that a key can be removed from a leaf without leaving it empty. There are three ways 1-key nodes (except the root) are eliminated.

1. When remove() encounters a 1-key node (except the root), it tries to steal a key from an adjacent sibling. But we can't just steal the sibling's key without violating the search tree invariant. This figure shows remove's "rotation" action, when it reaches "30". We move a key from the sibling to the parent, and we move a key from the parent to the 1-key node. We also move a subtree S from the sibling to the 1-key node (now a 2-key node).

   Note that we can't steal a key from a non-adjacent sibling.

```
    -------                  -------
    |20 40|                  |20 50|
    -------                  -------
    /  |  \          =>     /   |   \
---- ---- ----------    ---- ------- -------
|10| |30| |50 51 52|    |10| |30 40| |51 52|
---- ---- ----------    ---- ------- -------
 /\   /\   / |  | \      /\   / | \   / | \
          S                          S
```

2. If no adjacent sibling has more than one key, a rotation can't be used. In this case, the 1-key node steals a key from its parent. Since the parent was previously treated the same way (unless it's the root), it has at least two keys, and can spare one. The sibling is also absorbed, and the 1-key node becomes a 3-key node. The figure illustrates remove's action when it reaches "10". This is called a "fusion" operation.

```
    -------               ----
    |20 40|               |40|
    -------               ----
    /  |  \     =>         /  \
---- ---- ----    ---------- ----
|10| |30| |50|    |10 20 30| |50|
---- ---- ----    ---------- ----
```

3. If the parent is the root and contains only one key, and the sibling contains only one key, then the current 1-key node, its 1-key sibling, and the 1-key root are merged into one 3-key node that serves as the new root. The depth of the tree decreases by one.

   Eventually we reach a leaf. After processing the leaf, it has at least two keys (if there are at least two keys in the tree), so we can delete the key and still have one key in the leaf.

For example, suppose we remove 40 from the large tree depicted earlier. The root node contains 40, which we mark "xx" here to remind us that we plan to replace it with the smallest key in the root node's right subtree. To find that key, we move on to the 1-key node labeled "50". Following our rules for 1-key nodes, we merge 50 with its sibling and parent to create a new 3-key root labeled "20 xx 50".

```
                |20 xx 50|
            /-----------------\
        /--/      /   \        \-----\
    ----      ----      ----          ----------
    |14|      |32|      |43|          |62 70 79|
    ----      ----      ----          ----------
    /  \      /  \      /  \          /   |  |   \
  ---- ---- ---- ---- ---- ---- ------- ---- ---- ----
  |10| |18| |25| |33| |42| |47| |57 60| |66| |74| |81|
  ---- ---- ---- ---- ---- ---- ------- ---- ---- ----
```

Next, we visit the node labeled 43. Again following our rules for 1-key nodes, we move 62 from a sibling to the root, and move 50 from the root to the node containing 43.

```
                 |20 xx 62|
            /--------------------\
      /----/    /       \         \-----\
  ----      ----      -------            -------
  |14|      |32|      |43 50|            |70 79|
  ----      ----      -------            -------
  /  \      /  \     /    |   \          /   |  \
 ---- ---- ---- ---- ---- ---- ------- ---- ---- ----
 |10| |18| |25| |33| |42| |47| |57 60| |66| |74| |81|
 ---- ---- ---- ---- ---- ---- ------- ---- ---- ----
```

Finally, we move down to the node labeled 42. A different rule for 1-key nodes requires us to merge the nodes labeled 42 and 47 into a 3-key node, stealing 43 from the parent node.

```
                   |20 xx 62|
             /--------------------\
       /----/       /  \          \-----\
   ----      -------/    \------          -------
   |14|      |32|           |50|          |70 79|
   ----      ----           ----          -------
   /  \      /  \           /  \          /  |  \
---- ---- ---- ---- ---------- ------- ---- ---- ----
|10| |18| |25| |33| |42 43 47| |57 60| |66| |74| |81|
---- ---- ---- ---- ---------- ------- ---- ---- ----
```

The last step is to remove 42 from the leaf node and replace "xx" with 42.

```
                   |20 42 62|
             /--------------------\
       /----/       /  \          \-----\
   ----      -------/    \------          -------
   |14|      |32|           |50|          |70 79|
   ----      ----           ----          -------
   /  \      /  \           /  \          /  |  \
---- ---- ---- ----    ------- ------- ---- ---- ----
|10| |18| |25| |33|    |43 47| |57 60| |66| |74| |81|
---- ---- ---- ----    ------- ------- ---- ---- ----
```

3.3 Running times

A 2-3-4 tree with depth d has between 2^d and 4^d leaf nodes. If n is the total number of nodes in the tree, then $n \geq 2^(d + 1) - 1$. By taking the logarithm of both sides, we find that d is in O(log n).

The time spent visiting a 2-3-4 node is typically longer than in a binary search tree (because the nodes and the rotation and fusion operations are complicated), but the time per node is still in O(1).

The number of nodes visited is proportional to the depth of the tree. Hence, the running times of the find(), insert(), and remove() operations are in O(d) and hence in O(log n), even in the worst case.