CS 61B: Lecture 26
Monday, March 31, 2014

Today's reading: Goodrich & Tamassia, Section 10.1.

Representing Binary Trees

Recall that a binary tree is a rooted tree wherein no node has more than two children. Additionally, every child is either a _left_child_ or a _right_child_ of its parent, even if it is its parent's only child.

In the most popular binary tree representation, each tree node has three references to neighboring tree nodes: a "parent" reference, and "left" and "right" references for the two children. (For some algorithms, the "parent" references are unnecessary.) Each node also has an "item" reference.

```
public class BinaryTreeNode {          |  public class BinaryTree {
  Object item;                         |    BinaryTreeNode root;
  BinaryTreeNode parent;               |    int size;
  BinaryTreeNode left;                 |  }
  BinaryTreeNode right;                |

  public void inorder() {
    if (left != null) {
      left.inorder();
    }
    this.visit();
    if (right != null) {
      right.inorder();
    }
  }
}
```

```
===============================================
+ BINARY TREE | -------------------           +
============== |---          --- |            +
+              ||.|root  size|7| |            +
+              |-+-          --- |            +
+              --|----------------            +
+                v  BinaryTree object         +
+              -----                          +
+              | * |                          +
+              -----             ------------ +
+ Root node => |add|             |  parent  | +
+              -----             ------------ +
+              |.|.|             |   item   | +
+              /---\             ------------ +
+             /  ^^ \            |left|right| +
+            v  /  \ v           ------------ +
+           ---/-  -\---         structure of +
+           | . |  | . |      BinaryTreeNodes +
+           -----  -----                      +
+           |sub|  |div|                      +
+           -----  -----                      +
+          >|.|.|  |.|.|<                     +
+         / /--|-  -|--\ \                    +
+        / /  ^|    |^  \ \                   +
+       / v   |v    v|   v \                  +
+     --+--  --+--  --+--  --+--              +
+     | . |  | . |  | . |  | . |              +
+     -----  -----  -----  -----              +
+     | 6 |  | 5 |  | 9 |  | 3 |              +
+     -----  -----  -----  -----              +
+     |*|*|  |*|*|  |*|*|  |*|*|              +
+     -----  -----  -----  -----              +
===============================================
```

BINARY SEARCH TREES

An _ordered_dictionary_ is a dictionary in which the keys have a total order, just like in a heap. You can insert, find, and remove entries, just as with a hash table. But unlike a hash table, you can quickly find the entry with minimum or maximum key, or the entry nearest another entry in the total order. An ordered dictionary does anything a dictionary or binary heap can do and more, albeit more slowly.

```
A simple implementation of an ordered dictionary is a binary search tree,
                   wherein entries are maintained in a (somewhat) sorted order.
       18          The _left_subtree_ of a node is the subtree rooted at the
      /  \         node's left child; the _right_subtree_ is defined similarly.
    12    25       A binary search tree satisfies the _binary_search_tree_
   / \    / \      _invariant_:  for any node X, every key in the left subtree
  4  15  25  30    of X is less than or equal to X's key, and every key in the
 /  /  \    /      right subtree of X is greater than or equal to X's key.  You
1  13  17  28      can verify this in the search tree at left:  for instance,
 \  \       \      the root is 18, its left subtree (rooted at 12) contains
  3  14      29    numbers from 1 to 17, and its right subtree (rooted at 25)
                   contains numbers from 25 to 30.
```

When a node has only one child, that child is either a left child or a right child, depending on whether its key is smaller or larger than its parent's key. (A key equal to the parent's key can go into either subtree.)

An inorder traversal of a binary search tree visits the nodes in sorted order. In this sense, a search tree maintains a sorted list of entries. However, operations on a search tree are usually more efficient than the same operations on a sorted linked list.

Let's replace the "Object item;" declaration in each node with "Entry entry;" where each Entry object stores a key and an associated value. The keys implement the Comparable interface, and the key.compareTo() method induces a total order on the keys (e.g. alphabetical or numerical order).

[1] Entry find(Object k);

```
public Entry find(Object k) {
  BinaryTreeNode node = root;                     // Start at the root.
  while (node != null) {
    int comp = ((Comparable) k).compareTo(node.entry.key());
    if (comp < 0) {                               // Repeatedly compare search
      node = node.left;                           // key k with current node; if
    } else if (comp > 0) {                        // k is smaller, go to the left
      node = node.right;                          // child; if k is larger, go to
    } else {    /* The keys are equal */          // the right child.  Stop when
      return node.entry;                          // we find a match (success;
    }                                             // return the entry) or reach
  }                                               // a null pointer (failure;
  return null;                                    // return null).
}
```

This method only finds exact matches. What if we want to find the smallest key greater than or equal to k, or the largest key less than or equal to k? Fortunately, when searching downward through the tree for a key k that is not in the tree, we are certain to encounter both
  - the node containing the smallest key greater than k (if any key is greater)
  - the node containing the largest key less than k (if any key is less).
See Footnote 1 for an explanation why.

```
      +--+
      |18|
      /--\--+
    12   |25|
   / \   +/-\--+
  4  15  25 |30|
 /  /  \  +-/+-+
1  13  17 |28|
 \  \     +-\+
  3  14      29
```

For instance, suppose we search for the key 27 in the tree at left. Along the way, we encounter the keys 25 and 28, which are the keys closest to 27 (below and above).

Here's how to implement a method smallestKeyNotSmaller(k): search for the key k in the tree, just like in find(). As you go down the tree, keep track of the smallest key not smaller than k that you've encountered so far. If you find the key k, you can return it immediately. If you reach a null pointer, return the best key you found on the path. You can implement largestKeyNotLarger(k) symmetrically.

```
[2]  Entry min();
     Entry max();
```

min() is very simple. If the tree is empty, return null. Otherwise, start at the root. Repeatedly go to the left child until you reach a node with no left child. That node has the minimum key.

max() is the same, except that you repeatedly go to the right child. In the tree above, observe the locations of the minimum (1) and maximum (30) keys.

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

insert() starts by following the same path through the tree as find(). (find() works _because_ it follows the same path as insert().) When it reaches a null reference, replace that null with a reference to a new node storing the entry (k, v).

Duplicate keys are allowed. If insert() finds a node that already has the key k, it puts the new entry in the left subtree of the older one.
(We could just as easily choose the right subtree; it doesn't matter.)

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

remove() is the most difficult operation. First, find a node with key k using the same algorithm as find(). Return null if k is not in the tree; otherwise, let n be the first node with key k.

If n has no children, we easily detach it from its parent and throw it away.

If n has one child, move n's child up to take n's place. n's parent becomes the parent of n's child, and n's child becomes the child of n's parent. Dispose of n.

If n has two children, however, we have to be a bit more clever. Let x be the node in n's right subtree with the smallest key. Remove x; since x has the minimum key in the subtree, x has no left child and is easily removed. Finally, replace n's entry with x's entry. x has the key closest to k that isn't smaller than k, so the binary search tree invariant still holds.

```
        18                          18                             18
       /  \                        /  \                           /  \
     12    25                    12    25                       12    25
    / \    / \                  / \    / \                     / \    / \
   4  15  25  30 -insert(2)->  4  15  25  30 -remove(30)->    4  15  25  28
  /  /  \    /                /  /  \    /                   /  /  \      \
 1  13  17  28               1  13  17  28                  1  13  17      29
  \  \       \                \  \       \                   \  \
   3  14      29               3  14      29                  3  14
                              /                              /
                             2                              2
```

```
                                  18                    18
                              +--/  \                  /  \
                              |12|   25              13    25
                              /-\+   / \            / \    / \
       -remove(12)->   4  15  25  28   ->   4  15  25  28
                     /+-/+ \       \       /  /  \      \
                    1 |13| 17      29    1  14  17      29
                     \+-\+                \
                      3  14                3
                     /                    /
                    2                    2
```

To ensure you understand the binary search tree operations, especially remove(), I recommend you inspect Goodrich and Tamassia's code on page 446. Be aware that Goodrich and Tamassia use sentinel nodes for the leaves of their binary trees; I think these waste an unjustifiably large amount of space.

Running Times of Binary Search Tree Operations

```
     o
    / \
   o   o
  /\   /\
 o o   o o
/\ /\ /\ /\
oo oo oo oo
```

In a perfectly balanced binary tree (left) with height h, the number of nodes n is 2^(h+1) - 1. (See Footnote 2.) Therefore, no node has depth greater than log_2 n. The running times of find(), insert(), and remove() are all proportional to the depth of the last node encountered, so they all run in O(log n) worst-case time on a perfectly balanced tree.

On the other hand, it's easy to form a severely imbalanced tree like the one at right, wherein these operations will usually take linear time.

```
1
 \
  2
   \
    3
     \
      4
       \
        5
         \
          6
```

There's a vast middle ground of binary trees that are reasonably well-balanced, albeit certainly not perfectly balanced, for which search tree operations will run in O(log n) time. You may need to resort to experiment to determine whether any particular application will use binary search trees in a way that tends to generate somewhat balanced trees or not. If you create a binary search trees by inserting keys in a randomly chosen order, or if the keys are generated by a random process from the same distribution, then with high probability the tree will have height O(log n), and operations on the tree will take O(log n) time.

Unfortunately, there are occasions where you might fill a tree with entries that happen to be already sorted. In this circumstance, you'll create the disastrously imbalanced tree depicted at right. Technically, all operations on binary search trees have Theta(n) worst-case running time.

For this reason, researchers have developed a variety of algorithms for keeping search trees balanced. Prominent examples include 2-3-4 trees (which we'll cover next lecture), splay trees (in one month), and B-trees (in CS 186).

===

Footnote 1: When we search for a key k not in the binary search tree, why are we guaranteed to encounter the two keys that bracket it? Let x be the smallest key in the tree greater than k. Because k and x are "adjacent" keys, the result of comparing k with any other key y in the tree is the same as comparing x with y. Hence, find(k) will follow exactly the same path as find(x) until it reaches x. (After that, it may continue downward.) The same argument applies to the largest key less than k.

Footnote 2: A perfectly balanced binary tree has 2^i nodes at depth i, where 0 <= i <= h. Hence, the total number of nodes is $\sum_{i=0}^{h} 2^i = 2^{h+1} - 1$.