Binary Trees

What are Trees

- When you think of a tree the most common occurance that we see on a daily basis is our file directories, where we have subdirectires, and files in a tree like structure.
- A binary search tree is a binary tree with one more restriction: All children to the left of a node have smaller values, whereas all children to the right will have larger values.

- If a tree is balanced the average search time is directly proportialn to its height: O(h). In the above example the longes path from the root node to a leaf is three, so you can expect the average search time of O(3).
- Remeber that constants don't matter so the average seach time could be reduced to O(1).

Minimum

- The minimum of a binary search tree is the node with the smallest value, and, thanks to the properties of a binary search tree, finding the minimum couldn’t be simpler. Simply follow the left links starting from the root of the tree to find the one with the smallest value. In other words, the minimum is the leftmost node in the tree.

Maximum

- Whereas the minimum of a binary search tree is the node with the smallest value, the maximum is the node with the largest value. Finding the maximum is very similar to finding the minimum except that you follow the right links instead of the left. In other words, the maximum is the rightmost node in the tree.

Successor

- A node’s successor is the one with the next largest value in the tree. For example, given the tree shown:
    - The successor of A is D

    - The successor of H is I
    - The successor of I is K.

- Finding the successor is not that difficult, but it involves two distinct cases.

- In the first case, if a node has a right child, then the successor is the minimum of that. For example, to find the successor of I, we see that it has a right child, L, so we take its minimum, K. The same holds for the letter L: It has a right child, M, so we find its minimum, which in this case happens to be M itself.

- Conversely, if the node has no right child—as is the case with H—you need to search back up the tree until you find the first “right-hand turn.” By this we mean that you keep looking up the tree until you find a node that is the left child, and then use its parent. In this example, you move up the tree moving left (that is, following

Predecessor

- The predecessor of a node is the one with the next smallest value. For example, the predecessor of P is M, the predecessor of F is D, and the predecessor of I is H.
- The algorithm for finding the predecessor is essentially the inverse of what you’d use for the successor, and it involves two similar cases. In the first case, if a node has a left child, then you take its maximum. In the second case—whereby the node has no left child—you work up the tree until you find a “lefthand” turn.

Seaching

- When searching for a value in a binary search tree, you start at the root node and follow links left or right as appropriate until either you find the value you are looking for or there are no more links to follow. This can be summarized in the following steps:
    - Start at the root node.
    - If there is no current node, the search value was not found and you are done. Otherwise, proceed to step 3.
    - Compare the search value with the key for the current node.
    - If the keys are equal, then you have found the search key and are done. Otherwise, proceed to step 5.
    - If the search key sorts lower than the key for the current node, then follow the left link and go to step 2.
    - Otherwise, the search key must sort higher than the key for the current node, so follow the right link and go to step 2.

Seaching example

- In the below example it shows how you would search for the letter K in the binary search tree.
- Starting with the root node (step 1), you compare your search value, K, with the letter I
- Because K comes before I, you follow the right link (step 6), which leads you to the node containing the letter L
- You still don't have a match, but because K is smaller than L, you follow the left link (step 5).
- Finally, you have a match: The search value is the same as the value at the current node (step 4), and your search completes. You searched a tree containing nine values and found the one you were looking for in only three comparisons. In addition, note that you found the match three levels down in the tree—O(h).
- Each time you move down the tree, you effectively discard half the values—just as you do when performing a binary search of a sorted list. In fact, given a sorted list, you can easily construct the equivalent binary search tree.

Insertion

- Insertion is nearly identical to searching except that when the value doesn't exist, it is added to the tree as a leaf node. In the previous search example, if you had wanted to insert the value J, you would have followed the left link from K and discovered that there were no more nodes. Therefore, you could safely add the J as the left child of K.
- The newly inserted value was added as a leaf, which in this case hasn't affected the height of the tree.
- Inserting relatively random data usually enables the tree to maintain its O(log N) height, but what happens when you insert nonrandom data such as a word list from a dictionary or names from a telephone directory?
- Can you imagine what would happen if you started with an empty tree and inserted the following values in alphabetical order: A, D, F, H, I, K, L, M, and P?
- Considering that new values are always inserted as leaf nodes, and remembering that all larger values become right children of their parent, inserting the values in ascending order leads to a severely unbalanced tree.

Balancing

- The order in which data is inserted into and deleted from binary search trees affects the performance. More specifically, inserting and deleting ordered data can cause the tree to become unbalanced and, in the worst case, degenerate into a simple linked list. As mentioned, you can use balancing as a means of restoring the desired characteristics of the tree.
- One of the most difficult tasks in maintaining a balanced tree is detecting an imbalance in the first place. Imagine a tree with hundreds if not thousands of nodes. How, after performing a deletion or insertion, can you detect an imbalance without traversing the entire tree?
- Two Russian mathematicians, G. M. Adel'son-Vel'skii and E. M. Landis (hence the name AVL), realized that one very simple way to keep a binary search tree balanced is to track the height of each subtree. If two siblings ever differ in height by more than one, the tree has become unbalanced.
- Here is a tree that needs rebalancing. Notice that the root node's children differ in height by two.
- After an imbalance has been detected, you need to correct it, but how? The solution involves rotating nodes to remove the imbalance. You perform this rebalancing by working up the tree from the inserted/ deleted node to the root, rotating nodes as necessary anytime a node is inserted or deleted from an AVL tree.
- There are four different types of rotation depending on the nature of the imbalance: a single rotation and a double rotation, each with a left and right version. The table shows you how to determine whether a single or a double rotation is required.

Inbalanced	Child Is Balanced	Child Is Left-Heavy	Child Is Right-Heavy
Left-Heavy	Once	Once	Twice
Right-Heavy	Once	Twice	Once

The Unit Test

The NodeTest class defines some instance variables—one for each node shown above and initializes them in setUp() for use by the test cases. The first four nodes are all leaf nodes and as

such need only the value. The remaining nodes all have left and/or right children, which are passed in as the second and third constructor parameters, respectively:

```
@Before
public void setUp(){
    _a = new Node("A");
    _h = new Node("H");
    _k = new Node("K");
    _p = new Node("P");
    _f = new Node("F", null, _h);
    _m = new Node("M", null, _p);
    _d = new Node("D", _a, _f);
    _l = new Node("L", _k, _m);
    _i = new Node("I", _d, _l);
}
```

The design calls for the **minimum()** and **maximum()** methods (among others) to be part of the Node class. This enables you to find the minimum and maximum of a tree by querying the root node. It also makes testing much easier. The methods testMinimum() and testMaximimum() are pretty straightforward:

You simply ensure that each node in the tree returns the correct value as its minimum or maximum, respectively:

```
@Test
public void testMinimum() {
    assertSame(_a, _a.minimum());
    assertSame(_a, _d.minimum());
    assertSame(_f, _f.minimum());
    assertSame(_h, _h.minimum());
    assertSame(_a, _i.minimum());
    assertSame(_k, _k.minimum());
    assertSame(_k, _l.minimum());
    assertSame(_m, _m.minimum());
    assertSame(_p, _p.minimum());
}

@Test
public void testMaximum() {
    assertSame(_a, _a.maximum());
    assertSame(_h, _d.maximum());
    assertSame(_h, _f.maximum());
    assertSame(_h, _h.maximum());
    assertSame(_p, _i.maximum());
    assertSame(_k, _k.maximum());
    assertSame(_p, _l.maximum());
    assertSame(_p, _m.maximum());
    assertSame(_p, _p.maximum());
}
```

Next are successor() and predecessor(). Again, you put these methods on Node, rather than have them as utility methods in BinarySearchTree.

The method testSuccessor(), for example, confirms that the successor for “A” is “D”, for “D” is “F”, and so on, just as in the earlier examples. Notice that because “A” has no predecessor and “P” no successor, you expect the result to be null in both cases:

```
@Test
public void testSuccessor() {
    assertSame(_d, _a.successor());
    assertSame(_f, _d.successor());
    assertSame(_h, _f.successor());
    assertSame(_i, _h.successor());
    assertSame(_k, _i.successor());
    assertSame(_l, _k.successor());
    assertSame(_m, _l.successor());
    assertSame(_p, _m.successor());
    assertNull(_p.successor());
}

@Test
public void testPredecessor() {
    assertNull(_a.predecessor());
    assertSame(_a, _d.predecessor());
    assertSame(_d, _f.predecessor());
    assertSame(_f, _h.predecessor());
    assertSame(_h, _i.predecessor());
    assertSame(_i, _k.predecessor());
    assertSame(_k, _l.predecessor());
    assertSame(_l, _m.predecessor());
    assertSame(_m, _p.predecessor());
}
```

You also create another pair of tests—testIsSmaller() and testIsLarger()—for methods that have thus far not been mentioned but come in very handy later. A node is considered to be the smaller child if it is the left child of its parent. Conversely, a node is considered to be the larger child only if it’s the right child of its parent:

```
public void testIsSmaller() {
    assertTrue(_a.isSmaller());
    assertTrue(_d.isSmaller());
    assertFalse(_f.isSmaller());
    assertFalse(_h.isSmaller());
    assertFalse(_i.isSmaller());
    assertTrue(_k.isSmaller());
    assertFalse(_l.isSmaller());
    assertFalse(_m.isSmaller());
    assertFalse(_p.isSmaller());
}

@Test
public void testIsLarger() {
    assertFalse(_a.isLarger());
    assertFalse(_d.isLarger());
    assertTrue(_f.isLarger());
    assertTrue(_h.isLarger());
    assertFalse(_i.isLarger());
    assertFalse(_k.isLarger());
    assertTrue(_l.isLarger());
    assertTrue(_m.isLarger());
    assertTrue(_p.isLarger());
}
```

Finally, you create some tests for equals(). The equals() method will be very important when it comes time to test the BinarySearchTree class, as it enables you to compare the structure produced when inserting and deleting nodes with the expected result. The implementation will start from the current node and compare the values as well as the left and right children all the way down to the leaf nodes.

In testEquals(), you construct a replica of the node structure. You then compare each of the instance variables with their local variable counterparts, as well as check some boundary conditions just to make sure you haven’t hard-coded equals() to always return true!

```
@Test
public void testEquals() {
    Node a = new Node("A");
    Node h = new Node("H");
    Node k = new Node("K");
    Node p = new Node("P");
    Node f = new Node("F", null, h);
    Node m = new Node("M", null, p);
    Node d = new Node("D", a, f);
    Node l = new Node("L", k, m);
    Node i = new Node("I", d, l);
    assertEquals(a, _a);
    assertEquals(d, _d);
    assertEquals(f, _f);
    assertEquals(h, _h);
    assertEquals(i, _i);
    assertEquals(k, _k);
    assertEquals(l, _l);
    assertEquals(m, _m);
    assertEquals(p, _p);
    assertFalse(_i.equals(null));
    assertFalse(_f.equals(_d));
}
```

Performance

- Performance Comparison for 1,000 Inserts into a Binary Search Tree

Insertion Type	Comparisons*
Random Insertion	11,624
In-order Insertion	499,500