CS242 - Notes for Lab 1

1. The height of the tree is the length of the path from the root to the most distant child. A tree with no nodes has a height of -1.
2. insertAtLeaf does a normal BST insertion at the leaf. You may generally assume that there will be no duplicate keys, but if there are, put them in the left subtree.
3. smallInOrder creates a string of the node keys as they would appear in an inorder traversal. The second parameter in inOrderList provides the delimiter that separates each key name. The last key name is also followed by a delimiter.
4. The charsAtWidest function finds the number of characters the tree is across at its widest. The second parameter to the function provides the space between nodes. This computation is best produced with a breadth-first search.
5. The findLongestKey function returns the length of the longest key in the tree.
6. The printTree function is already written, and will work once you have written the supporting functions that already part of the assignment.
7. The toString function provides a string that is generated from a breadth-first traversal of the tree.
	1. Each time the traversal moves to the next level it inserts a newline (“\n”).
	2. Each time the traversal encounters a null-link it inserts a 0.
	3. The second parameter in toString provides the delimiter that separates each node.
8. insertAtRoot uses the algorithm described in Sedgewick – the same algorithm you used to complete your first homework assignment.
9. Although the functions written so far do not record the number of children each node has, some of the functions that we write next will require this information. Add a private data member to the BinaryTree class that records the number of children the node has – this number includes all the descendants of this node. The countChildren() function will set that private data member to the correct value.
10. selectNodeWithNSmaller(tree, n) returns a node that contains (n+1)th smallest key. For example,
	1. selectNodeWithNSmaller(tree, 0) returns a node that contains the smallest key
	2. selectNodeWithNSmaller(tree, 1) returns a node that contains the 2nd smallest key
	3. selectNodeWithNSmaller(tree, 2) returns a node that contains the 3rd smallest key
	4. selectNodeWithNSmaller(tree, 0) returns a node that contains the 4th smallest key
11. partitionTest(tree, n) moves the node that contains the (n+1)th smallest key to the root.
12. balance(tree) uses the partition function to generate a balanced tree. Don’t forget that partition does not necessarily conserve the child count, so you will need to call countChildren after each partition, or modify partition to preserve the child count.