**CMSC 206: Data Structures**

**Lab #11: Binary Search (Trees)**

**Part I. Binary Search**

1. In a new Java class, write this method, based tightly on what we did in class:

/\*\* Search for an int in a sorted array using binary search

\* Pre-condition: the array is in non-decreasing order

\* @param array The non-decreasing array

\* @param target The number you're searching for

\* @return The index of target, or -1 if target is not in

\* the array. If target appears multiple times in

\* the array, chooses a valid index for target

\* arbitrarily.

\*/

public static int binarySearch(int[] array, int target)

1. We saw an implementation of binary search in class. Test this implementation by making a main method that creates a sorted array and calls binarySearch. Make sure to test for corner cases, like an empty array (e.g., made with new int[0]), searching for elements not in the array, and searching for elements at the beginning and end of the array.
2. Now, you will rewrite binarySearch recursively. Here's a start:

/\*\* Search for an int in a sorted array using binary search

\* (identical in behavior to binarySearch)

\*/

public static int binarySearchR(int[] array, int target)

{

return binarySearchRec(array, target, 0, array.length);

}

/\*\* Recursively search for an int in a portion of a sorted

\* array using binary search.

\* Pre-conditions: the array is in non-decreasing order

\* 0 <= lo <= hi <= array.length

\* @param array The array to search in

\* @param target The number you're searching for

\* @param lo The first index in the portion of the array to

\* search in

\* @param hi One past the last index in the portion of the

\* array to search in

\* @return The index of target, or -1 if target is not found

\*/

public static int binarySearchRec(int[] array, int target,

int lo, int hi)

The idea here is that the binarySearchRec method replaces the action of the loop in the iterative version. At any given recurrence, the lo and hi parameters bound the region of the array to look in. These numbers will successively get closer and closer as the binary search proceeds.

1. (Optional) Generalize the binarySearch method by having it work on an array of any type E that implements Comparable<E>. Here is the signature:

public static <E extends Comparable<E>>

int binarySearch(E[] array, E target)

The <E extends Comparable<E>> bit declares that E is a type variable standing in for a type that is a subtype of Comparable<E>. You are not expected to know or be able to produce this syntax, but the implementation of binarySearch in terms of compareTo should be straightforward.

**Part II. Binary Search Trees**

1. Choose 8 words and add them to a binary search tree, on paper.
2. Write down preorder, inorder, and postorder traversals of that tree.
3. Check your work by downloading the *BinarySearchTree.java* implementation (with the *Node.java* and *BMCSet.java* files) from the syllabus page and adding those strings to the binary search tree. Use the Eclipse debugger to see what nodes are connected to what. Is your drawing correct?
4. Write printPreOrder and printPostOrder methods in BinarySearchTree. These should be analogous to printInOrder (at the bottom of the file) but print different traversals. Both will need recursive helper methods.
5. Modify the addRec method to swap the cmp < 0 and cmp > 0 conditions. This will make the implementation incorrect. Think about how this will change the way trees are built. Draw a picture of a tree that is built with this addRec.
6. Demonstrate that this modified addRec is wrong by writing a main method that adds several items to a tree but then cannot find them with contains.

**Part III. A Binary Search Tree Iterator**

Iterating through a binary search tree is a challenge. I have an implementation of this iterator that you will explore below.

1. Download the *StackInt.java* and *SingleLinkedList.java* files from the syllabus page. (Make sure that the version of *SingleLinkedList.java* you downloaded indeed implements StackInt.)
2. Download the *BSTIterator.java* from the syllabus.
3. Modify your BinarySearchTree to implement the Iterable interface, writing the iterator method to construct a BSTIterator.
4. You should now be able to iterate through a BinarySearchTree, doing an inorder traversal. Confirm this by using an iterator to print out all elements in a BinarySearchTree that you create in main.
5. The code in BSTIterator is not simple. To try to understand it, trace the behavior of the iterator's stack field as you iterate through the tree you drew earlier. Before running any code in Eclipse, draw the state of the stack as it changes every time next() is called. Then, run your program in the debugger, and watch to see whether your drawings are correct.
6. Can you come up with better descriptions of goLeft and goRight?