In this homework assignment, you will re-implement the ApptBook ADT (with some changes) using a binary search tree (BST) data structure. A BST permits an efficient lookup mechanism (as opposed to linear search) so there is an efficiency test to ensure your code can handle half a million entries.

1 The Binary Search Tree Data Structure

Please read section 9.5 in the textbook for a description of the binary search tree data structure. Alternatively, there are many web-pages/lecture notes on BST, the wiki page may be a good starting point (https://en.wikipedia.org/wiki/Binary_search_tree). In the textbook (as well as some online sources), a separate BTNode class is used; we will not do that. Use a nested node class as before.

Linked lists and arrays support insertion, removal or finding an entry in time proportional to the number of entries in the container. Trees, on the other hand, offer a significant efficiency advantage in that they generally support these operations in time proportional to the \( \log \) of the number of entries (assuming the tree is kept balanced).

In order to achieve the potential time efficiency of binary search trees, one needs to keep the tree roughly balanced. In CompSci 535, you will learn some of the techniques used to do this (as it happens, the tree-based containers in Java’s libraries use “red-black” trees). But in this course, we will ignore the problems of unbalanced trees. Do not make any attempt to re-balance your tree. The efficiency tests we run will make sure to construct a balanced tree.

2 Concerning the Modified ApptBook ADT

As with previous assignments, the appointment book ADT includes appointments in sorted order. We are making two changes in the ADT (as well as changing the implementation!):

- Duplicate insertions are ignored.
- Removal is not supported.

Previously an appointment could be inserted multiple times into the appointment book. As it happens, handling duplicates in a binary search is very tricky to get right. (The obvious solutions fail in subtle ways when the tree is reorganized, for example after deletion.) Rather than have you implement something with subtle errors, we have decided to say that inserting an appointment into an appointment book when it is already there will have no effect. This matches normal uses of appointment books and calendar applications.

For this homework, we do not support removal. That is in order to reduce the complexity of the implementation. Removal will be added in the following homework assignment.
The efficiency tests check to see that you built the tree correctly. If you wrote the code efficiently, this test shouldn’t take more than 15 seconds or so. If you wrote the inefficient (but easy) technique, the test will take much longer and you will lose points.

3 Private Recursive Helper Methods

The implementation will make use of several (recursive) helper methods:

**isInProperOrder(r,lo,hi)** Check that all nodes in the subtree rooted at “r” have valid (non-null) data that are in the exclusive range given. The bounds (“lo” or “hi”) can be null which means no bound. So if “lo” is null, it means that there is no lower bound for the data in the subtree. The null subtree satisfies any bounds since it has no nodes in it.

**countNodes(r)** Return the number of nodes in the given subtree. A null subtree has zero nodes.

**isInSubtree(r,n)** Return true if the node “n” can be found within the subtree “r.” (“Found” means that we use the order to look for the node rather than looking everywhere.)

**insertAllHelper(r)** Insert all data from all nodes in the subtree rooted by “r’ into this appointment book.

**cloneSubtree(r,c)** Make a copy of the subtree rooted at “r” and set this book’s cursor if the node being copied is the cursor (“c”).

The first three helper methods are useful for the invariant checker `wellFormed` and the last two are useful for `insertAll` and `clone` respectively.

4 What you need to do

You need to declare the Node class with a constructor that takes an appointment. Then you need to complete the implementations of the first three helper methods and then the well-formedness checker. Then you can test the invariant checker.

Then you need to implement the rest of the methods in the class with the exception of `removeCurrent`. The `advance` method will use the technique of “Repeated Search” as explained in the Navigating Trees handout.

We provide a modified test case for the ADT and a new efficiency test.