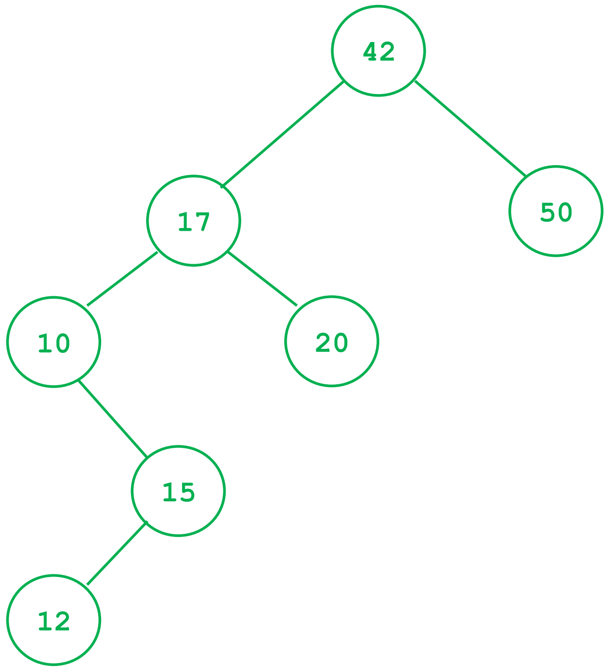
CSE 143 Final Part 1, August 18, 2011 **Sample Solution**

Question 1. (16 points) Binary Search Trees. (a) Draw a picture that shows the integer binary search tree that results when the following numbers are inserted into a new, empty binary search tree in the order given:

42 17 10 50 15 20 12



(b) Write down the order in which the nodes in your tree would be visited using the following tree traversal orders:

preorder **42 17 10 15 12 20 50**

postorder **12 15 10 20 17 50 42**

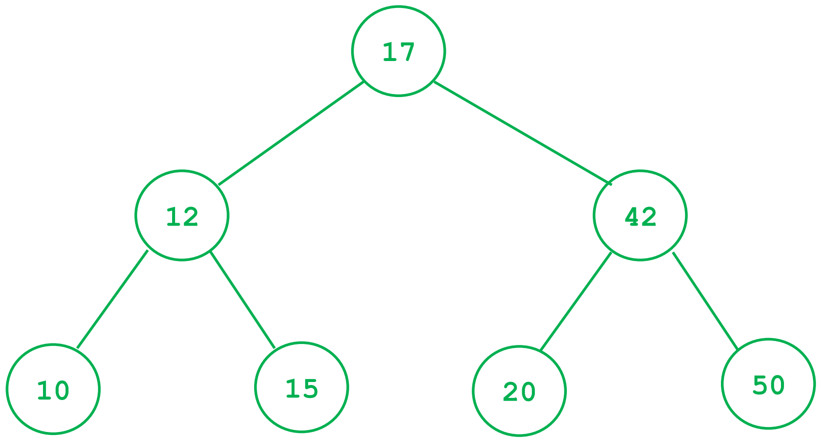
inorder **10 12 15 17 20 42 50**

Note: When grading part (b), answers that corresponded correctly to the tree drawn in part (a) received full credit, even if there were errors in the part (a) tree.

Question 2. (8 points) Binary Search Trees. Draw a picture that shows an integer binary search *with* ***minimum height*** that contains the following numbers. If there is more than one possible way to store the nodes in a binary search tree with minimum height, then you can show any of the possible solutions.

42 17 10 50 15 20 12

(Note: This is exactly the same data as in the previous problem, except that this time you're being asked to construct a minimum height binary search tree that contains the numbers. The numbers can be added to the tree in any order, not necessarily the one given. Hint: in a minimum-height tree, both subtrees of each interior node will have approximately the same number of nodes.)



Question 3. (20 points) Tree search. Suppose we have a class that stores a set of integer values with no duplicates. The class uses a binary search tree internally to store the values.

For this problem, complete the definition of method `numLessThan` below so it returns the number of items in the set whose value is less than the method's argument `n`. You may define additional auxiliary methods if you find them useful. The next page is blank to provide additional space for your answer if needed. You may define as many additional simple variables as you wish, but no additional collections, arrays, or tree nodes, and you may not make any changes to the tree itself.

For full credit, your solution must take advantage of the binary search tree organization and not examine any more of the tree than necessary.

```
public class IntTreeSet {
   // tree nodes
   private class IntTreeNode {
      public int val;              // data value in this node
      public IntTreeNode left;     // left subtree with values < val
      public IntTreeNode right;    // right subtree with values > val
   }

   // binary search tree storing contents of this IntTreeSet
   private IntTreeNode treeRoot;

   // construct new empty IntTreeSet
   public IntTreeSet() {
      treeRoot = null;
   }

   // code for other methods omitted to save space...

   // return number of items in this set with value < n
   public int numLessThan(int n) {
      return numLessThan(treeRoot, n);
   }

   // return number of nodes in tree with root r that have a value < n
   private int numLessThan(IntTreeNode r, int n) {
      if (r == null) {
         return 0;
      }
      if (r.val >= n) {
         // any values < n are in the left subtree only
         return numLessThan(r.left, n);
      } else {
         // r.val < n.  Result is 1 (this node) + counts in subtrees
         return 1 + numLessThan(r.left, n) + numLessThan(r.right, n);
      }
   }
```

Question 4. (20 points) Collections. Complete the following method so it will scan a list of words, find all of the words in that list that do not appear in a dictionary, and record how many times those words not in the dictionary appear in the original list.

The input parameters to the method are a List (words) of strings and a Set (dictionary) of strings. The method should return a Map containing <String,Integer> pairs, where the String in each pair is a word that appears in the input list but not in the dictionary, and the associated integer is the number of times the corresponding word appeared in the list. The entries in the resulting Map do not need to be sorted in any particular order. You may create additional collections if you need them. String values must match exactly to be considered to be the same: i.e., "foo", "Foo", "FOO" and " FOO   " are all different.

```
// Return a Map containing a list of <String,Integer> pairs where
// each String is a String that appears in the List words but not
// in the Set dictionary, and the associated Integer is the number
// of times the word appears in the List words

Map<String,Integer> countUnknownWords(List<String> words,
                                      Set<String> dictionary) {
   // allocate map to hold result frequencies
   Map<String,Integer> freq = new HashMap<String,Integer>();

   // process words in the dictionary
   for (String word: words) {
      if (!dictionary.contains(word)) {
         // input word is not in the dictionary.  count it
         if (freq.containsKey(word)) {  // or (freq.get(word)!=null)
            // increase count of existing word
            freq.put(word, freq.get(word)+1);
         } else {
            // add new word
            freq.put(word,1);
         }
      }
   }

   // return result map
   return freq;
}
```

Notes: We accepted solutions that used `for (i = ...) { ... get(i) ... }` to access the elements in `words`, even though this could be potentially expensive if `get(i)` is not a constant-time operation for the particular `List` implementation used. The problem should have been worded to disallow this, but since we missed that we did not deduct anything. Solutions that explicitly used iterator `hasNext()` and `next()` methods were, of course, fine.

We also should have restricted the problem to limit the number of extra collections. The map used to accumulate the result is necessary, but some solutions did things like allocate extra stacks and queues and make copies of the entire input list, which is not needed. No points were deducted for solutions that did this.

Any suitable `Map` implementation like `HashMap` or `TreeMap` could be used to accumulate the result.

Question 5. (16 points) Class design. For an address book application we have created the following class to represent the name and address of each entry in the address book.

```
public class AddressCard implements Comparable<AddressCard> {
   // instance data
   private String firstName;
   private String lastName;
   private String city;
   private int zipCode;

   // Create new AddressCard with given data
   public AddressCard(String fn, String ln, String city, int zip) {
      firstName = fn;  lastName = ln; this.city = city; zipCode = zip;
   }

   // add any additional code needed below

   // compare this AddressCard to other
   public int compareTo(AddressCard other) {
      int comp = this.lastName.compareTo(other.lastName);
      if (comp == 0) {
         comp = this.firstName.compareTo(other.firstName);
      }
      return comp;
   }
}
```

We would like to modify this class so we can compare one `AddressCard` object to another. If `x` and `y` are `AddressCard` objects, then `x.compareTo(y)` should compare `x` to `y` and return an appropriate integer depending on the ordering of `x` and `y`. `AddressCards` are ordered by their `lastName` fields. If two cards have the same `lastNames`, then the `firstNames` are used to decide the ordering. The `lastName` and `firstName` strings should be compared using their natural ordering as Strings and should not be modified. In particular, two strings that are similar but differ only in capitalization ("Abc" and "abc") are not equal.

Make the necessary modifications and additions to the above class to allow `AddressCard` objects to be compared.

Notes: The use of "`this`" in `this.firstName` and `this.lastName` is, of course, optional.

Question 6. (10 points) Abstract classes and interfaces. (Brief, to-the-point answers are appreciated)

In addition to regular classes, both abstract classes and interfaces can be used in Java programs to define new types.

(a) Describe one advantage of defining a new type using an abstract class instead of an interface.

Abstract classes can contain implementations of some or all methods, which can then be inherited by subclasses. Interfaces are pure type specifications and cannot include methods, so classes that implement interfaces have to provide implementations for all of the methods specified in the interface, or inherit suitable implementations from a superclass.

(b) Describe one advantage of defining a new type using an interface instead of an abstract class.

Classes can implement more than one interface, allowing a class to implement multiple types that are not necessarily related by inheritance. Classes cannot inherit from more than one superclass, whether or not it is abstract.