Policy Change Because we are going to end up with only 4 programming exercises instead of the 8 promised, you will be allowed to turn in for full credit any of the four exercises as long as you have your final version completed by the end of finals week. Obviously you will want to complete Exercise 4, which will give you practice working with binary trees, before the final exam time.

With this new policy, you did not need to submit a "serious attempt" by any particular time—you should only submit work that you believe is complete, for feedback, or specific questions about your work (and please make clear which type of email it is so I can try to respond earlier to questions).

Trees

A *tree* is a data structure composed of a number of *nodes*, where each node has zero or more arrows (references) leading from it to other nodes. In addition, each node stores a single data item.

Trees have a lot of associated terminology (some of which we've already used informally when we studied the heap data structure): the single node that has no arrows pointing to it is the *root* node of the tree; a node that is pointed to by another node is known as a *child* of the node that points to it, and the node that points to it is known as the *parent* node; a node with no children is a *leaf* node; a path from the root node to a leaf node is known as a *branch*; the height of a tree is the length of the longest branch; a tree where every node has at most two children is known as a *binary tree* and the two children are known as the *left child* and the *right child*; each node can be viewed as being the root of its own *subtree* consisting of itself and all its children, and their children, and so on, for all the *descendants* of the node.

We will only study binary trees (except for material at the end of the course). Earlier we drew pictures of binary trees implemented in an array, but for our current work, we will only look at trees implemented using linked-list-like techniques, with a node being something like (where we use `String` data for simplicity):

```
public class Node
{
  private String data;          // holds the data for the node
  private Node left, right;  // holds arrows pointing to left and
                             // right child nodes
}
```

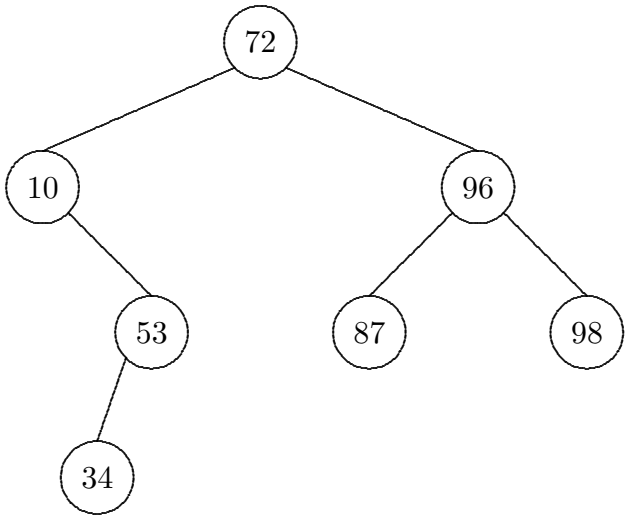
Further, we will focus on using a binary tree with a special property as yet another way to implement the `Sack` ADT.

Binary Search Trees

We will now look at yet another way to implement the sack ADT, using a binary tree that has a special property. Recall that we have already seen the idea of a binary tree satisfying some special property—the heap property, but here the property is different.

Given a binary tree where the data items in the nodes are comparable (we will use `String` or `int` data items for simplicity), we say that the binary tree has the *binary search tree property* (really we just say that it *is* a binary search tree) if for every node, all of the data items in the left subtree are less than the node's data item and all of the data items in the right subtree are greater than or equal to the node's data item.

Here is a simple example of a binary search tree:



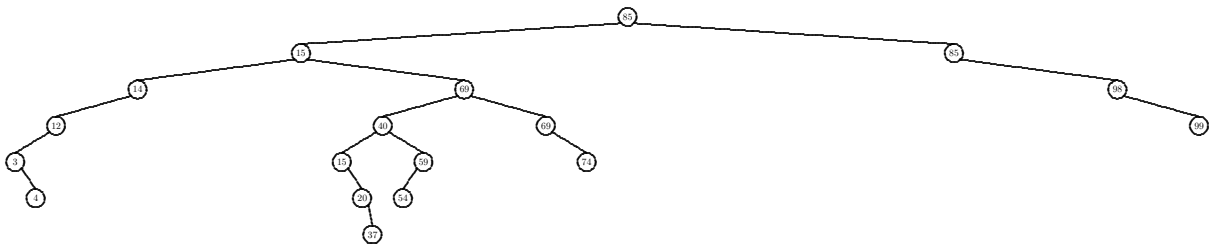
Adding an Item to a Binary Search Tree

The idea for adding an item to a binary search tree involves recursive thinking, as follows. If the item is less than the root item, then we recursively add the item to the left subtree, and otherwise we add the item to the right subtree. If the subtree we want to add to is empty, we simply make the root node point to a new node containing the item being added.

If we perform this algorithm on these numbers, namely

85, 15, 69, 14, 69, 12, 40, 15, 3, 20, 85, 98, 4, 74, 59, 37, 99, 54,

we obtain this binary search tree:



⇐ Add the items in order and verify that this tree is the correct binary search tree.

⇐ Given a binary search tree with n items, what is the worst-case time efficiency for adding an item? What is the best-case time efficiency for adding an item?

⇐ Begin development of the binary search tree implementation of the sack ADT—we'll call this class `SackBST`—by sketching out the instance variables, the constructor, and the `add` method.

Finding an Item in a Binary Search Tree

One of the crucial operations for implementing the sack ADT is the `find` method. To find an item—actually a key, but we are treating keys as items, really—is beautifully simple and recursive: starting from some node, we compare the target key to that node's data. If the target is less than the data, we recursively search the left subtree, if the target is greater than the data, we recursively search the right subtree, and if the target is the same as the data, then we have found it.

⇐ Try to implement the `find` method in the `SackBST` class.

⇐ Try to implement the `get` method using the stored information from a successful `find`.

⇐ Think about the `remove` method—but we'll discuss this a lot more later, and you'll have to actually implement it in Exercise 5, and we'll probably have to improve the `find` method.

Traversing a Binary Search Tree

Any implementation of the sack ADT must have the ability to traverse the items stored in the sack, namely to visit them one at a time in whatever order is convenient for the machinery. Binary trees have a number of reasonable traversals. We might think of doing breadth-first traversal—go level by level, left to right—but this turns out to be a hard way to traverse a binary tree that is stored using nodes and pointers (it was the easiest way for a complete binary tree stored in an array in that order). There are three much easier traversals that use the recursive nature of the binary tree.

The in-order traversal is the one that is most useful for us here. The idea of this traversal is to recursively do in-order traversal of the left subtree, visit the root node, and then recursively do in-order traversal of the right subtree.

For our first binary search tree example, the in-order traversal says, ask your assistant to do in-order traversal of the tree with 10 as its root. That person visits the items, following the same process, in the order 10, 34, 53. Then you visit 72 and ask an assistant to traverse the right subtree, which produces 87, 96, 98.

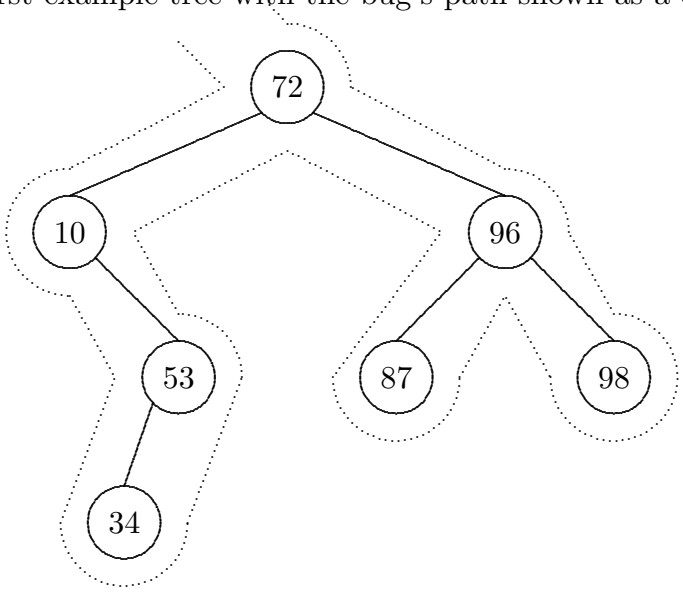
⇐ Using people in the class as workers, do the in-order traversal of the bigger binary tree, and note that we have stumbled on yet another sorting algorithm!

> We can also traverse a binary tree in pre-order: visit the root item, then recursively do pre-order traversal of the left subtree, and then recursively do pre-order traversal of the right subtree, and post-order: post-order traverse the left subtree, post-order traverse the right subtree, and visit the root item.

⇐ Perform these traversals on the big example tree.

If we picture a bug (or other smallish animal) starting at the root of a binary tree and keeping its left paw on the tree and walking all the way around the tree, we note that it encounters each typical node three times: first when it comes to the node from its parent, second when it returns to the node from its left child, and third when it returns to the node from its right child (if it is missing the left and/or right child nodes, we can still count going nowhere and coming back).

Note that if we want to literally follow this path to produce one of these traversals of the tree, we need to give a node a pointer to its parent, in addition to its pointers to its left and right children.
Here is our first example tree with the bug's path shown as a dashed curve:



Given this path around the tree, we note that the in-order traversal is just the nodes listed in the order of second contact, the pre-order traversal is the nodes listed in the order of first contact, and the post-order traversal is the nodes listed in the order of third contact.

The Binary Search Tree Sorting Algorithm

This sorting algorithm is to take the list of items to be sorted and add them to a binary search tree, and then perform the in-order traversal of that binary tree to produce the sorted items.

The worst-case time efficiency of this sorting algorithm depends on the height of the binary search tree that is produced as the items are added. It can vary between $O(n^2)$ and $O(n \log n)$).

⇐ Describe how n items could be arranged so that the resulting binary search tree would have $\log n$ levels, and n levels.
