Lab 8 -- Postponed Lab 7 on Binary Search Trees

The goal of this lab is to design and implement a linked-tree structure that uses a union data type.

Binary Search Trees

A tree, much like a list, is made of several nodes. An empty node does not contain any data, nor references to other nodes. A non-empty node contains one piece of data with references to other nodes. No node in the tree is referred to by more than one node. This structure results in a tree structure. If a non-empty node refers to at most two other nodes, it is a *binary* tree. In this case, these two nodes are often called the left and right children, following how such a tree is usually drawn on paper.

a

/ \

b c

/ \ \

d e f

In the above illustration, the node *a* is called the root of the tree. Node *b* is the root of the left subtree of *a* , while node *c* is the root of the right subtree of *a*. As you can see, a tree structure is naturally recursive (i.e., a tree is made of smaller trees).

A special kind of binary tree is a *binary search tree*. This type of tree enforces an additional condition on its nodes, based on the ordering of the data they keep. In a binary search tree, for any non-empty node, the data at that node is *strictly greater* than all the data in the nodes in its left subtree, and is *strictly less* than all the data in its right subtree. This is called the binary search property, and must be obeyed by **every** non-empty node in a binary search tree.

Using numbers as data in nodes, here is an example of a binary search tree.

7

/ \

3 10

/ / \

1 8 12

\

9

As its name suggests, such a tree is good at searching.

**Tree Traversals**

Given a tree, it is possible to traverse the tree in several ways. Two primary ways are depth-first and breadth-first traversals.

In a depth-first traversal we choose a direction (left or right) and keep digging until we reach the end of the road. Then we come up one step and dig in the other direction, and so on. Furthermore, we may process before, between, or after digging in the two child trees. This creates a total of 6 possible depth-first traversals. Fixing the order in which we choose the two child trees as "first left, then right," we have 3 traversals, characterized as follows:

* **Pre-order traversal**: process node, traverse left, traverse right
* **In-order traversal**: traverse left, process node, traverse right
* **Post-order traversal**: traverse left, traverse right, process node

The other 3 possibilities would be the above, but swapping "left" and "right."

**Count the height of the tree**

A leaf node is defined as a node that does not have any children or data. The leaf nodes are not pictured in the illustration above. The height of a node is defined as one less than the maximum number of nodes on a path from that node to a descendant leaf (the height of a leaf is 0). The height of the tree is the height of its root.

7

/ \

3 10

/ / \

1 8 12

\

9

In the above example, the heights of 1, 9 and 12 are 1. The height of 8 is 2, the height of 10 is 3 and the height of 7 is 4 (same as the height of the tree).

7

/ \

3 10

/ \ / \

1 x 8 12

/ \ / \ / \

x x x 9 x x

/ \

x x

For reference, the same tree, but with leaf nodes now pictured, denoted by x.

What to do

**Package:** bst

Download [this interface for a binary search tree](https://northeastern.instructure.com/courses/63372/files/8174406?wrap=1)[download](https://northeastern.instructure.com/courses/63372/files/8174406/download?download_frd=1)and carefully read through the documentation for each method. In this assignment, you are asked to design and implement a *recursive union* that implements the functionality of the binary search tree.  Similar to the recursive lists, most binary search tree operations exploit the recursive nature of binary search trees. The binary search tree property helps us to argue how each operation in a tree can be defined as corresponding to similar operations on subtrees. Take a moment to *design* a solution!

Once you are ready, implement the above interface in a BinarySearchTreeImpl class. In addition to the methods listed in the interface, you will need to implement the toString method. Similar to lists, you should implement node classes for the tree and implement the functionality in terms of node operations. This class should also have a constructor that creates an empty tree.