Module 7 Lab -- Binary Search Trees

Due Sunday by 11:59pm **Points** 100

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

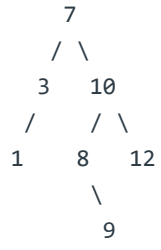
Binary Search Trees

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.

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.



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.

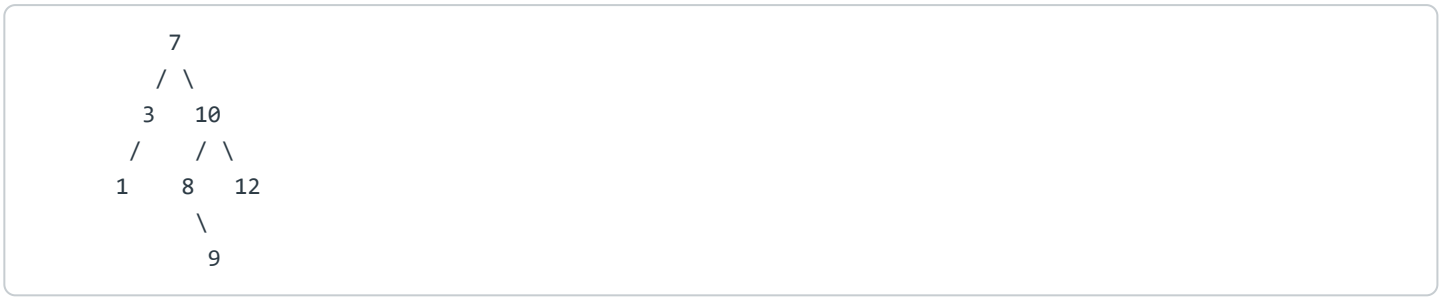
In a depth-first traversal we choose a direction (left or right) and keep digging until we reach the end of the road. Then we come up one step and dig in the other direction, and so on. Furthermore, we may process before, between, or after digging in the two child trees. This creates a total of 6 possible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

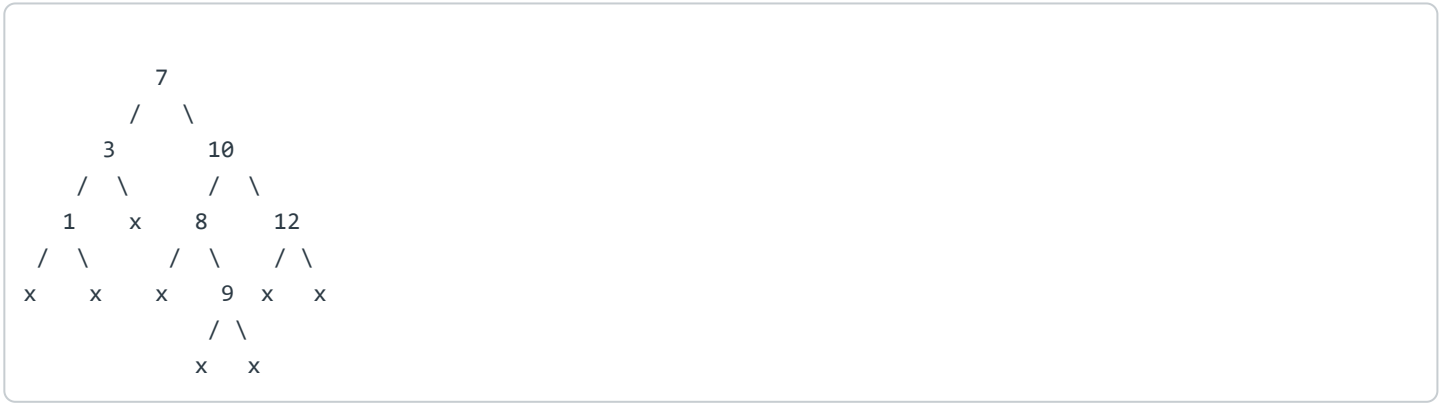
The other 3 possibilities would be the above, but swapping "left" and "right."

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.



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).



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

What to do

Package: bst

Define and implement the the interface `BST` for a binary search tree that contains the following methods:

- `public void add(T obj)` : insert an object in the tree. If the object is already present in the tree, does nothing.
- `public int size()` : return the size of this tree (i.e. the number of elements in this tree).
- `public boolean present(T obj)` : returns `true` if this object is present in the tree, `false` otherwise.
- `public T minimum()` : returns the smallest object (defined by the ordering) in the tree, and `null` if the tree is empty.
- `public T maximum()` : returns the largest object (defined by the ordering) in the tree, and `null` if the tree is empty.
- Override `toString` to return a String of the form `[1 2 3]` where 1, 2, and 3 are the elements of the tree, in ascending order.
- Write three methods -- `preOrder`, `inOrder`, `postOrder` -- that traverse the tree in the three orders described above. Each of them should return a string with the data in the node in the order in which they process them, separated by spaces, in the form `[1 2 3]` (i.e. "process node" means creating and returning a string-representation of the data in the node).
- Write a method `height` that will compute and return the height of the the tree.

Similar to 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. Try to formulate insertion and membership operations as recursive operations on a tree.

Write an implement `BSTImpl` for the above interface. 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.

Testing

Whenever you write a class, you should also write tests for that class that proves not only that your code CAN work but that it WILL ALWAYS work. Your goal here is to write tests to achieve as close to 100% coverage as possible. But even more importantly, your tests should be sufficient to ***convince someone else that your code works correctly***.

What to Submit

1. Create a zip file that directly contains only your src/ and test/ folders. When you unzip the file, you must see only these two folders.
2. Log on to the **Handins server (https://handins.ccs.neu.edu/)**
3. Navigate to this lab and submit the zip file.
4. Wait for a few minutes for the grader's feedback to appear, and take action if needed.

Grading Criteria

This assignment will be assessed via the automatic testing available on the Handins server. You should ensure that your code does not have any code style violations and that it passes all of the test cases. This is worth 1% of your final grade.

Additionally, your code will be assess via peer evaluation. Each student will be required to review one other student's code for an additional 0.5% of their final grade. During the peer review your code will be evaluated based on three criteria:

1. Design and organization of classes, interfaces, and methods including
    - How appropriately you captured the data and operations in the implementation
    - Whether code looks well-structured and clean (not unnecessarily complicating things or using unwieldy logic)
    - How well your design and implementation embraces various design principles that you have learned so far
2. Understandability and quality of documentation including
    - Whether the documentation enhances the understandability of the code rather than just repeating how something is implemented
    - Whether it is stand alone, meaning whether the documentation is sufficient for use without reading the implementation
3. Quality and coverage of tests including
    - Whether tests are modular -- do you write one big, monolithic test method or does your test class have separate methods for each test
    - Whether tests are written in such a way so that the reader knows what is being tested
    - Whether the tests convince the reader that if all tests pass then the code it is testing works correctly