ASSIGNMENT 9 – GENERICS

REQUIRED FILES

```
./BinarySearchTree.java
./PublicTests.java
./it5001/collections/immutable/
        ImmutableEmptyList.class
        ImmutableList.class
        ImmutableListIterator.class
        ImmutableListNode.class
./doc/
        index.html
        A bunch of files...
```

PREAMBLE

Parameterized types are great; they allow us generalize over types and not need to write excessive boilerplate code while maintaining type safety of our program.

In this assignment, you will complete the implementation of a Binary Search Tree found in `BinarySearchTree.java`. Some of the methods require the implementation of `ImmutableList<T>` we discussed in Lecture 19; the compiled byte code and supporting documentation have been provided to you.

BINARY SEARCH TREES

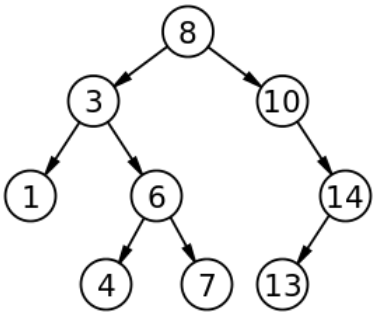

Fig 1. *A binary search tree of integers* (image source: Wikipedia)

"In computer science, a binary tree is a widely used abstract data type that represents a hierarchical tree structure with a set of connected nodes. Each node in the tree can be connected to up to two children, but must be connected to exactly one parent, except for the root node, which has no parent. These constraints mean there are no cycles or "loops" (no node can be its own ancestor), and that each child can be treated like the root node of its own subtree, making recursion a useful technique for tree traversal.

A Binary Search Tree (BST) is a rooted binary tree data structure with the value of each internal node being greater than all the values in the respective node's left subtree and less than (or equal to) the ones in its right subtree." – Wikipedia

The wonderful part about BSTs is not just that operations are dependent of the height of the tree (making a lot of them in $O(\log n)$ average complexity), but in each node, the left and right children are both BSTs themselves, making BSTs a very simple data structure to implement.

RESOURCES PROVIDED

BINARYSEARCHTREE.JAVA

`BinarySearchTree.java` is an incomplete implementation of a simple mutable BST which we will complete in this assignment. Three methods are unimplemented. Most of the infrastructure has already been created. The attributes of a BST are:

- A root value of type `T`
- A left child (who is itself a BST) of type `BinarySearchTree<T>`
- A right child (who is itself a BST) of type `BinarySearchTree<T>`

There are also two constructors: 1) a public constructor that creates a BST of some root value, and 2) a private constructor that creates an empty BST. There are also two methods which you may need to use at some point in this assignment:

- `add(T t)` adds an element to the BST. If the BST is empty, it simply assigns the element as the root value, and initializes its children as empty BSTs. Otherwise, if `t` is smaller than the root value, then we add `t` (recursively) to the left child, otherwise add `t` (recursively) to the right child.
- `addAll(Iterable<? extends T> it)` adds all the elements in it to this BST. An `Iterable` is simply something that can be used in a for loop (just like Python iterables). We set the type argument to be `? extends T` because we only need the iterable to be a producer of `T`.

Most importantly, `T` must be a subtype of `Comparable<? super T>` because we want to assert that the elements of the BST must be comparable, i.e. we must be able to compare two instances of `T` so we know which child to add/search an element into/for. `Comparable` is an interface that provides a `compareTo` method; read up on the `Comparable` interface for more details (you may see how we compare two values of `T` in the add method in our BST implementation). This is why in the header of our BST class declaration, the declaration of type parameter `T` is `T extends Comparable<? super T>`.

IMMUTABLELIST IMPLEMENTATION

Some of the methods we will implement require some resizeable list type. We know that arrays are of fixed size which makes them unsuitable; therefore we have provided a compiled version of the `ImmutableList` implementation for your use. This implementation is much more robust and contains many more methods, though admittedly you won't need to use most of it. We only provided compiled java bytecode since you do not need to read the source code of the implementation. To understand how to use it, you may simply refer to the documentation for the `ImmutableList` interface in `doc/index.html`. The interface has already been imported in the `BinarySearchTree.java` file, so you can make use of the interface directly.

You are also not allowed to decompile and edit the source code (you can decompile it but that won't help you very much). At the end of the assignment, you will only submit the `BinarySearchTree.java` file, so editing the `ImmutableList` source code won't help you.

RUNNING TESTS

Coursemology has pretty poor support for customized Java testing, therefore we have provided our own test infrastructure to you. `PublicTests.java` can be compiled and executed together with your `BinarySearchTree` class and the provided `ImmutableList` implementation. Simply execute the `PublicTests` class to see detailed information of each public test case.

To ensure that the infrastructure has been set up properly, start by being in the root of the assignment folder (containing `BinarySearchTree.java`, `PublicTests.java`, `doc/` and `it5001/`), then run the following commands:

```
javac *.java
java PublicTests
```

You should see that your code compiles but fails all public test cases initially.

PART 1: CONTAINS [30 MARKS]

Complete the implementation for the `public boolean contains(T t)` method in the `BinarySearchTree` class, which determines whether some object `t` is in the BST. The logic for this method is relatively simple:

1. If this tree is empty, obviously return `false`;
2. If the root value of this tree is equal to `t`, obviously return `true`;
3. Thanks to BSTs being recursive data structures, we can determine where `t` might be if it is not the root:
    a. If `t` < the root value, if it exists in the BST it must be in the left child, so search there (recursively)
    b. Otherwise, if it exists in the BST it must be in the right child, so search there (recursively)

Notes:

- The BST is empty if and only if the root value is equal to `null`
- When checking if the root value is equal to `t`, use the `equals()` method attached to all `Objects` (`==` is similar to `is` in Python; it works for `null`, but for all other objects it's advisable to use `equals` to check for equality)
- `a.compareTo(b)` returns `-1` if `a < b`, `0` if `a == b` and `1` otherwise

To check your solution, compile and test your code. Start by being in the root of the assignment folder (containing `BinarySearchTree.java`, `PublicTests.java`, `doc/` and `it5001/`), then run the following commands:

```
javac *.java
java PublicTests
```

The two public test cases for this part are Test 1 and Test 2.

PART 2: INORDER TRAVERSAL [40 MARKS]

Tree traversal is a means of walking through each element in a binary tree. We are going to focus on in-order traversal. The in-order traversal of a BST is to perform an in-order traversal of the left subtree, then to the root, then to the in-order traversal of the right subtree (notice that once again this is recursive). For example, the in-order traversal of the BST in Figure 1 would be

`1 : 3 : 4 : 6 : 7 : 8 : 10 : 13 : 14`

Complete the implementation of the `public ImmutableList<T> inorder()` method. The method produces the in-order traversal of the BST as an `ImmutableList`. Once again, the logic for this method is relatively simple:

1. If the tree is empty, obviously return an empty list;
2. Otherwise, obtain the in-order traversal of the left subtree (recursively), append the root value, and concatenate that with the in-order traversal of the right subtree (recursively);

Notes:

- See `empty()` in the `ImmutableList` interface;
- See `concat`, `prepended` and `appended` in the `ImmutableList` interface;

To check your solution, compile and test your code. Start by being in the root of the assignment folder (containing `BinarySearchTree.java`, `PublicTests.java`, `doc/` and `it5001/`), then run the following commands:

```
javac *.java
java PublicTests
```

The two public test cases for this part are Test 3 and Test 4.

PART 3: TREESORT [30 MARKS]

If you were observant, you would notice that the in-order traversal of a BST is a sorted list! Therefore, we can make use of BSTs to sort any sequence of elements.

Complete the implementation of the `public static <T extends Comparable<? super T>> ImmutableList<T> sorted(Iterable<? extends T> ls)` method. This method is `static`, and therefore is attached directly to the `BinarySearchTree` class. This method returns the sorted equivalent of some iterable `ls`, in the form of an `ImmutableList`. The logic for this method is again very simple:

1. Create an empty BST (the empty BST constructor is private but still accessible to the `BinarySearchTree` class)
2. Add all elements of the iterable into the empty BST
3. Return the BST's in-order traversal

Notes:

- See `addAll` in the `BinarySearchTree` class.
- You might want to use your `inorder` method to get the BST's in-order traversal

To check your solution, compile and test your code. Start by being in the root of the assignment folder (containing `BinarySearchTree.java`, `PublicTests.java`, `doc/` and `it5001/`), then run the following commands:

```
javac *.java
java PublicTests
```

The two public test cases for this part are Test 5 and Test 6.

SUBMISSION

- Upload **ONLY** your `BinarySearchTree.java` file to Coursemology.
- Ensure that it is named exactly as `BinarySearchTree.java`.
- **Do not zip your file**.
- Failure to do so will result in a large penalty since our script will not be able to find your file correctly.

GRADING

- You will be graded solely on private test cases.
- **If your code is not compilable, we cannot grade your submission.**
- The private test cases will not be made known to you, nor will you know in advance of the submission deadline whether you've passed/failed them. Generally, if you have followed the steps described in each part and passed all public test cases, you will be able to pass the private test cases too.
- After the submission deadline, we will compile your `BinarySearchTree.java` file together with the supporting files provided to you. Output logs of private testing will be added as a Coursemology comment in your submission for your review.
- Solutions will be provided after the submission deadline for review.