Assignment 1: ADTs and Generics

In this assignment, you will implement two implementations of a Map ADT in Java. The interface Map<K,V> is given in the provided file **Map.java**, which contains the interface definition together with Javadoc specifications for each of the methods. From this, it should be clear what each of these methods should do. Be aware that this interface is a much simplified version of the java.util.Map interface that is part of the Java API.

We have also provided two class implementation files, **BSTMap.java** and **HashTableMap.java** , which you will need to complete for this assignment. Both should be placed in a package called **csci235**, along with the provided Map.java file. If you fail to do this your code will not compile when we test it, and you will end up with zero points for the assignment.

Implementation 1: BSTMap<K, V>

This should use a simple binary search tree implementation whose nodes contain the paired keys and values, where the nodes themselves should be ordered via the keys. Note that you do not have to worry about balancing the tree or removing nodes.

We have provided the inner class definition for **Node** for your tree nodes. The class has been made private so that it can only be used inside of the BSTMap class, but we have made its fields public so they can be directly accessed and modified within BSTMap.

```java
private class Node {
    public K key;
    public V value;
    public Node left;
    public Node right;

    Node(K key, V value) {
        this.key = key;
        this.value = value;
    }
}
```

Another thing to note: Since the keys in a BST must have some ordering defined for them for properly organizing the nodes, we need to constrain the generic type K. We do this by using the **Comparable** interface provided by Java, which requires that an implementing class defines the compareTo() method with the following specification:

x.compareTo(y) returns:

- a negative integer, if x < y
- a positive integer, if x > y
- zero, if x equals y

Most basic object types like Integer, Character, and String implement this interface, and you can also create your own classes that implement this if you like. In any case, to constrain the type parameter to implement a particular interface, we use the extends (not implements!) keyword when we declare the type parameter in the < >'s. This is why the initial part of the class definition looks as follows:

```
public class BSTMap<K extends Comparable<K>, V> implements Map<K, V>  {
```

In the class definition, we have provided a field of type Node to reference the root of the tree, and a field called size to keep track of the number of key-value pairs in the map, which is the same as the number of nodes in the tree. We have also provided the constructor which initializes root to null and size to 0, as a new map should start with no key-value pairs.

For some of your method implementations, you might decide to use recursion instead of loops. In such cases, your "helper" method should be private, as it should not be usable outside of the class. Such an approach might look something like this:

```
public void method() {
     helperMethod(this.root);
}
private void helperMethod(Node node) {
      if (node == null) {
            // Maybe do something for the base case?
            return;
      }
      helperMethod(node.left);
      // Maybe do something with node.value
      helperMethod(node.right);
}
```

Implementation 2: HashTableMap<K, V>

This implementation of Map should use a hash table with separate chaining, where the number of buckets is doubled when the load factor reaches a fixed threshold of 4.0. As part of the provided code for HashTableMap, we have the inner class **KVPair**, which acts as a simple container for the key-value pairs in our implementation:

```
private class KVPair {
    public K key;
    public V value;

    KVPair(K key, V value) {
        this.key = key;
        this.value = value;
    }
}
```

Using this class, we have the following two fields for HashTableMap:

```
private List<KVPair>[] buckets;
private int size;
```

Here, we are using the java.util.List interface from the Java API for our individual buckets, which should simplify matters as List has many useful methods that can be called directly. We have also provided the code for the constructor to help set up the initial hash table Map. Note that we set each of the bucket entries in the array to new lists of key-value pairs, instead of keeping them null, so we do not have to worry about getting a NullPointerException any time we access one of the buckets.

For your convenience, we have provided a helper method that calls hashCode() for keys to determine the proper bucket index. Because Java has all object types implement hashCode() and equals() by default, we do not have to constrain the type of K in the initial part of the definition of HashTableMap, as we did with BSTMap. Simply typed objects such as Integer and String already have reasonable hashCode() and equals() implementations, so you shouldn't have to worry about using them as keys here in your implementation. If you decided to use your own custom type for the keys, be sure to provide proper and consistent overriding implementations of these two methods.

As was required with our hash table implementations in CSCI 152, when the current load factor meets or exceeds a fixed constant –in this case 4.0– you are required to create a new bucket array and rehash all of the key-value pairs currently in the map into this new bucket array, and then set the buckets field to this new array. Such an approach is necessary for us to achieve constant amortized time efficiency for large numbers of puts to the map. We have provided a method that calculates the current load factor for your use.

While it is not absolutely required, we do recommend that you implement the following helper function which attempts to retrieve a key-value pair with the given key from the given bucket, or returns null if no such pair exists:

```
private KVPair findPairInList(K key, List<KVPair> bucket) {
```

Many of your methods require that you try to find such a pair in a given bucket, so it should be easier to just write a single method that does this and reuse it.

Testing Your Implementations

To help get you starting in testing your implementations, we have provided a basic testing class for your implementations in **MapTester.java**. However, note that when we will do grading, we will use something more extensive, so be sure to add more tests to MapTester to more thoroughly validate your implementations before you submit them.

Submission

To receive credit for this assignment, you need to:

1. Submit your code to Moodle before the deadline. To do so properly, you need to find your project directory on your computer, and zip up the src folder, and submit the resulting src.zip file. Keep in mind that we will extract only your **BSTMap.java** and **HashTableMap.java** files for grading. For this reason, DO NOT modify the **Map.java** file, as we will be using the original version provided to you, and DO NOT change the names of the classes **BSTMap** and **HashTableMap**.
2. Do the assignment yourself! Don't use other's code, whether it is from online or from another student.
3. If you have questions about the assignment, please post your questions on Piazza. HOWEVER, never post code publicly! If you have problems that you feel requires that someone look at your code, post it "private to instructors" in Piazza.