Computer Science 331 — Winter 2017
Assignment #3

Instructions

This assignment concerns lecture material that was introduced in this course on or before Monday, March 13.

Please **read the entire assignment** before you begin work on any part of it.

This assignment is due by 11:59 pm on Monday, March 27.

There are a total of 100 marks available, plus an additional 25 bonus marks.

Questions

A *mapping* is a partial function $f : K \mapsto V$ from a set K of "keys" to a set V of "values." Since this is a partial function, there is *at most* one value $f(k) \in V$ that is defined for each key $k \in K$. If the set of keys k such that $f(k)$ is defined is finite, and there is a useful "linear order" defined for the set K, then a binary search tree is one of several data structures that can be used to represent the mapping f. Note that a mapping is another way to describe the dictionary ADT described in class.

The Java Collections Framework contains an interface `Map` describing such a mapping. The following questions will make use of the `SimpleMap` interface (available on the Assignment 3 web page), a simplified version of `Map`. The file `LinkedListMap.java` consists of an ordered linked list based implementation of `SimpleMap`. `CountWords.java` and `CountCompares.java` contain programs that use various implementations of a `SimpleMap`.

The purpose of this assignment is to give you practice working with various ways to implement a mapping. In particular, you will:

- create an implementation using a binary search tree
- create an implementation using a hash table with chaining
- create an implementation by adapting a class provided in the JCF called `TreeMap`, which implements a red-black tree.
- perform some numerical investigation on the performance of `search` using these implementations

1. **(10 marks)** Give pseudocode for an *iterative* algorithm for searching in a binary search tree.

2. **(25 marks)** Implement a Java class called `BSTMap` that implements the `SimpleMap` interface using a binary search tree. This class should satisfy the following:

   - The `search` method must be implemented *iteratively*. You must *not* use a stack to do this.
   - The `insert`, `delete`, and `modify` methods must be implemented *recursively*.
   - Includes a public method `int getNumSearch()` which returns the number of times that the `search` method has been called.
   - Includes a public method `int getNumCompares()` which returns the number of times that the `compareTo` method has been called inside of `Search`.

   Be sure to implement your class and the `search` method so that the required counts are maintained.

3. **(25 marks)** Implement a Java class called `ChainingHashMap` that implements the `SimpleMap` interface using a hash table with chaining. This class should satisfy the following:

   - The constructor should accept one parameter specifying the size of the hash table. The actual size should be the smallest prime number greater than or equal to this parameter. The `BigInteger` class contains methods for primality testing.
   - Use the *chaining* mechanism for collision resolution.
   - Use the *division* method for the hash function, and the default `hashCode` method of the key type.
   - Includes a public method `int getNumSearch()` which returns the number of times that the `search` method has been called.
   - Includes a public method `int getNumCompares()` which returns the number of times that the `compareTo` method has been called inside of `Search`.

   Be sure to implement your class and the `search` method so that the required counts are maintained.

4. **(10 marks)** Implement a Java class called `RBTMap` that implements the `SimpleMap` interface using a red-black tree. You should do this by using the `TreeMap` class from the Java API, and using existing functions of `TreeMap` to implement the required functions from the interface. The Java Collections API documentation, available through the course web page, provides all the information you need to know about `TreeMap` to enable you to complete this question.

   In addition, your class must include:

   - A public method `int getNumSearch()` which returns the number of times that the `search` method has been called.
   - A public method `int getNumCompares()` which returns the **theoretical upper bound** on the number of times that the `compareTo` method would have been called inside of `Search`.

Be sure to implement your class and the `search` method so that the required counts are maintained.

Note: The `CountWords.java` program will be used to test your `BSTMap`, `ChainingHashMap`, and `RBTMap` classes. This program accepts an optional 2nd command line parameter that allows you to choose which implementation of `SimpleMap` is used. As long as your classes correctly implement this interface, the only required modification to `CountWords.java` should be uncommenting the lines in the program where the appropriate `SimpleMap` instances are created. The TAs will use the version of this program provided on the web site as-is, and it is your responsibility to make sure that your classes work properly with it.

Note: Be sure to document your classes thoroughly using `javadoc` tags and assertions as appropriate. Both the quality of your implementation and documentation will be taken into account when grading this question. *10 of the 100 available marks for this assignment will be allocated to documentation.*

Note: Although you are not being asked to provide a test suite for this assignment, you are responsible for testing your classes as thoroughly as possible in order to make them as bug-free as possible. You should feel free to use JUnit or to add any auxiliary functions to your classes that will help with this process.

5. **(20 marks)** The `CountCompares.java` program constructs instance of `SimpleMap` using all of the above data structures, by inserting a sequence of random keys. It then performs a sequence of searches for random keys and computes the average number of calls to `compareTo` done per call to `search`. It accepts three command-line parameters:

   - `numKeys` — number of keys to insert in each `SimpleMap`
   - `numMaps` — number of `SimpleMaps` using each data structure to create, with `numKeys` keys in each
   - `sorted` — 1 if keys should be inserted in increasing order, random order otherwise

   The version of this program obtained from the Assignment 3 web page only uses the linked list implementation provided. As long as your classes from the previous questions correctly implement the `SortedMap` interface, the only required modification to `CountCompares.java` should be uncommenting certain lines in the program.

   Run `CountCompares.java`, modified to use all of the data structures from the previous questions that you were able to implement completely, using the following parameter values:

   (a) `numKeys` = 10000, `numMaps` = 100, `sorted` = 0
   (b) `numKeys` = 10000, `numMaps` = 100, `sorted` = 1

   Note that the computation will take a bit of time with these parameters!

   Present the data you obtain in a table. Is the data you obtained in the previous question what you would expect? Why or why not?

Your submission for this assignment should include the following files:

- `BSTMap.java` — binary search tree implementation
- `ChainingHashMap.java` — chaining hash table implementation
- `RBTMap.java` — adapter class using `TreeMap` to implement the `SortedMap` interface
- `MyRBTMap.java` — your solution to the bonus question (if attempted)
- `CountWords.java` and `CountCompares.java` — main programs, modified to use `LinkedListMap` and the classes that you have successfully implemented.
- A file containing answers to the written questions for Questions 1 and 5.

Bonus Questions

The following bonus question should only be attempted after the previous questions have been completed satisfactorily. While solutions to this question are not required for full credit on this assignment, correct solutions can result in a grade of more than 100 %.

6. **(25 bonus marks)** Implement a Java class called `MyRBTMap` that implements the `SimpleMap` interface using the red-black tree data structure from scratch, i.e., without using `TreeMap`. Modify the program `CountCompares.java` appropriately to use your class, and extend your data and comparisons from the last question to also use your class.