Assignment 9

Hash Tables: Separate Chaining

Introduction

In this assignment, you will implement a BST map and a hash map that both implement the Map ADT. The hash map will use Java's built-in List (ArrayList) as the underlying storage structure (this is the field named `table`).

Each entry in a map contains both a key and a value associated with that key. The elements in the hash map will be stored according to the hash code of the key and the size of the underlying table in the hash map. **NOTE:** The automated grading of your assignment will include some different and additional tests to those found in the `A7Tester.java` file. For all assignments, you are expected to write additional tests until you are convinced each method has full test coverage.

Objectives

Upon finishing this assignment, you should be able to:

- Describe all of the operations found in the `BSTMap`
- Write an implementation of the Hash Map ADT using separate chaining
- Compare and contrast the performance of different BST map implementations based on the performance measurement metrics provided

Submission and Grading

Attach **only** the `HashMap.java` file to the BrightSpace assignment page. Do not change or submit any other files. Remember to click **submit** afterward. You should receive a notification that your assignment was successfully submitted.

If you chose not to complete some of the methods required, you **must** provide a stub for the incomplete method(s) in order for our tester to compile. If you submit files that do not compile with our tester, you will receive a zero grade for the assignment. It is your responsibility to ensure you follow the specification and submit the correct files. Additionally, your code must not be written to specifically pass the test cases in the tester, instead, it must work on all valid inputs. We may change the input values during grading and we will inspect your code for hard-coded solutions.

Be sure you submit your assignment, not just save a draft. All late and incorrect submissions will be given a zero grade. A reminder that it is OK to talk about your assignment with your classmates, but not to share code electronically or visually (on a display screen or paper). Plagiarism detection software will be run on all submissions.

Part 1 Instructions

You have been provided with a working implementation of a Binary Search Tree and the BSTMap class that implements the Map ADT (specified in `Map.java`).

Examine the BST solutions and compare them with your Assignment 8 implementation. The implementation provided demonstrates a slightly different insert method than the one introduced in lecture. If you have any questions about the implementation, please post questions in the discussion board on BrightSpace.

Part 2 Instructions

The Hash Map implementation will make use of built-in Java generic List ADT to allow users of the hash map to specify the type for both the key and the value. Your hash map will use **separate chaining** to handle collisions, therefore the **table** field in `HashMap.java` is a List of Lists, where each sublist is a List of Entries: `private List< List< Entry<K,V> > > table;`

The constructor is provided for you. You will notice it initializes the table as a new `ArrayList` of the specified `tableSize`, and subsequently initializes every element in that table to a new empty `LinkedList`.

You will need to implement four methods: `containsKey`, `get`, `entryList` and `put`. For the methods that require you to search through the table and/or through a List within the table, we suggest you use the built-in Java `Iterator` class as shown during lecture.

Use the List `iterator` method to get an iterator for that list:

```
//gets the list at specified index in table
List<Entry<K,V» list = table.get(index);

// gets an Iterator that can iterate over the above list
Iterator<Entry<K,V» iter = list.iterator();
```

You can then call the Iterator methods to iterate over (traverse) the list.
Examples calls to useful iterator methods:
`iter.next()`
`iter.hasNext()`

See the Java API for full Iterator method documentation:
https://docs.oracle.com/javase/8/docs/api/java/util/Iterator.html.

Part 3 Instructions (Optional - not for marks, and not bonus)

We have provided you with a Map implementation using linked lists (called `LinkedMap.java`). In class, we discussed how it is that we expect the binary search tree implementation of Map and HashMap to perform less computation for the same tasks which use a list. You will be using the `BinarySearchTree.java` and `BSTMap.java` provided in this comparison.

In this assignment we will use a very simple approach to compare the three implementations: we will count the number of times loops execute in the get and put methods (or depth of the recursion depending on your implementation).

We have added four methods to support performance testing to the Map interface, and the implementation of those methods are given to you in LinkedMap.java and HashMap.java. You must add the code to your `HashMap.java` to support this performance testing as follows:

1. Look at the implementation of `LinkedMap.java` and `BinarySearchTree.java` to see how we counted the loop iterations. In particular, look at how the fields `getLoops` and `putLoops` are used. Change your `HashMap.java` to count the loops in the `put` and `get` methods.
2. Once you've done that, compile Performance.java and run it as follows:
   `java Performance linked 0`

   The results should look something like:

```
Doing initial tests:
Your solution should match exactly for linked and be comparable for BST and Hash.

-- Instructor's solution:
 [128 linked] put loop count: 499500
 [128 linked] get loop count: 1000000
 [128 bst   ] put loop count: 11079
 [128 bst   ] get loop count: 12952
 [128 hash  ] put loop count: 1
 [128 hash  ] get loop count: 0
--

--Your solution:
 [128 linked] put loop count: 499500
 [128 linked] get loop count: 1000000
 [128 bst] put loop count: 11079
 [128 bst] get loop count: 12952
 [128 hash] put loop count: 1
 [128 hash] get loop count: 0
--
```

   Once the loop counting is correct, perform some experiments described below to compare the different implementations and document the results in `performance.txt`.

If you run `java Performance linked 1000` it will gather information about the linked list implementation over 1000 put and gets. If you run `java Performance bst 1000` it will gather information about the binary search tree implementation over 1000 put and gets. If you run `java Performance hash 1000` it will gather information about the hash table implementation over 1000 put and gets.

In the supplied `performance.txt` file, you must copy and paste the results you obtain when running the various tests described in the file.

- You need to run the linked implementation lists of size 10, 100, 1000, 10000, 100000 and 300000
- You need to run the binary search tree implementation for trees of size 10, 100, 1000, 10000, 100000, 300000 and 1000000.
- You need to run the hash map implementation for hash maps of size 10, 100, 1000, 10000, 100000, 300000 and 1000000.

For example, here is the output produced by one run of the instructor's solution where there are 10000 insertions (or puts) of random values into the BST, and then 10000 searches (or gets) of random values from the BST:

```
> java Performance bst 10000
bst map over 10000 iterations.
[1489713840577 bst] put loop count: 146936
[1489713840577 bst] get loop count: 166935
```

The very large numbers represent the random seed used to generate random values for insertion into, and lookup from, the BST. The counts are the number of loop iterations performed in BST methods (i.e., insert and find) in order to implement the put and get in the Map (in this case a Map based on tree – i.e., the BST).

Similarly, here is the output produced by one run of the instructor's solution where there are 10000 puts of random values into the HashMap, and then 10000 gets of random values from the HashMap:

```
> java Performance hash 10000
hash map over 10000 iterations.
[1562196577084 hash] put loop count: 53
[1562196577084 hash] get loop count: 104
```

It is possible that your implementations will have slightly more or slightly fewer loops, but yours should be within a few percent of the instructor solution.