

Assignment 10

Reminder: Your code is to be designed and written by only you and not to be shared with anyone else. See the Course Outline for details explaining the policies on Academic Integrity. Submissions that violate the Academic Integrity policy will be forwarded directly to the Computer Science Academic Integrity Committee.

All materials provided to you for this work are copyrighted, these and all solutions you create for this work cannot be shared in any form (digital, printed or otherwise). Any violations of this will be investigated and reported to Academic Integrity.

Objectives

- Introduction to Hash Map ADT with separate chaining
- More practice with generics
- More practice with the Map ADT
- Introduction to performance measurement

Introduction

In this assignment you will implement a hash map that implements the Map ADT and uses Java's built in List/ArrayList as the underlying storage (table field). Each element in the map will contain 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.

Submission and Grading

Submit your `HashMap.java` with the completed methods through the assignment link in BrightSpace.

- You **must** name your methods as specified in the given interface and as used in `A10Tester.java` or you will receive a **zero grade** as the tester will not compile.
- If you chose not to complete some of the methods required, you **must at least provide a stub for the incomplete method** in order for our tester to compile.
- If you submit files that do not compile with our tester (ie. an incorrect filename, missing method, etc) you will receive a **zero grade** for the assignment.
- Your code must **not** be written to specifically pass the test cases in the testers, instead, it must work on other inputs. We may change the input values when we run the tests and we will inspect your code for hard-coded solutions.
- **ALL late and incorrect** submissions will be given a **ZERO** grade.

Getting Started

1. Read the comments in `HashMap.java` carefully and implement the methods until all tests pass in `A10Tester.java`

Part I - HashMap

Your hash map implementation will make use of built-in Java generic List ADT to allow users of the tree 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` is a List of Lists, where each sublist is a List of Entry instances:

```
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 the 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.

You can 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.

Example calls to useful iterator methods:

```
iter.next()
iter.hasNext()
```

See the Java API for full Iterator method documentation:

<https://docs.oracle.com/en/java/javase/16/docs/api/java.base/java/util/Iterator.html>

Part II – Performance Testing (optional – not bonus)

We have provided you with a Map implementation using linked lists (called `LinkedMap.java`). In class, we discussed the performance efficiencies of different data structures. It is expected that `BSTMap` and `HashMap` perform less computation for the same tasks than a list implementation. You will be using your `BinarySearchTree.java` and `BSTMap.java` from Assignment 9 in this comparison. Copy these files into your working Assignment 10 directory.

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 code to your `BinarySearchTree.java`, `BSTMap.java` and `HashMap.java` to support this performance testing as follows:

1) Replace the given `BinarySearchTree.java` and `BSTMap.java` with your completed versions from Assignment 9.

2) Update your `BinarySearchTree.java`

a. Add fields and methods for `findLoops` and `insertLoops` to

`BinarySearchTree.java` adding the code shown here:

```
long         findLoops;
long         insertLoops;

public BinarySearchTree () {
    root = null;
    count = 0;
    resetFindLoops();
    resetInsertLoops();
}

public long getFindLoopCount() {
    return findLoops;
}

public long getInsertLoopCount() {
    return insertLoops;
}

public void resetFindLoops() {
    findLoops = 0;
}

public void resetInsertLoops() {
    insertLoops = 0;
}
```

b. Look at the implementation of `LinkedMap.java` to see how we counted the loop iterations. In particular, look at how the fields `getLoops` and `putLoops` are updated in the `get` and `put` methods.

In the `insert` and `find` methods in your `BinarySearchTree.java` similarly update the fields `insertLoops` and `findLoops` so that they count loop iterations (or recursive calls).

3) Update `BSTMap.java`

a. Add the following methods to `BSTMap.java`

```
public long getGetLoopCount() {
    return bst.getFindLoopCount();
}

public long getPutLoopCount() {
    return bst.getInsertLoopCount();
}

public void resetGetLoops() {
    bst.resetFindLoops();
}

public void resetPutLoops() {
    bst.resetInsertLoops();
}
```

4) Update `HashMap.java`

- a. Again, look at the implementation of `LinkedMap.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.

Copyrighted Material

5) Once you have added all of the necessary code, compile and run `Performance.java` as follows:

```
javac Performance.java
java Performance linked 0
```

This will do 1000 insertions and 1000 lookups on each of the 3 ADTs.

You should see something resembling:

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
```

It is possible that your `BinarySearchTree` and `HashMap` implementations will have slightly more or slightly fewer loops. For example, your `put` loop count maybe approximately 1000 more than the instructor solution depending on your implementation (approximately 1 more loop iteration per insertion). Once you are satisfied your loop counting is correct, perform some experiments described in the next section to compare the two 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 `puts` and `gets`.

If you run:

```
java Performance hash 1000
```

it will gather information about the hash table implementation over 1000 `puts` and `gets`.

To get a sense of the runtime impact, run the following tests:

- Run the linked implementation lists of size 10, 100, 1000, 10000, 100000 and 300000
- Run the binary search tree implementation for trees of size 10, 100, 1000, 10000, 100000, 300000 and 1000000.
- 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
```