Implementing Multimaps

In programming, a *map* (a.k.a. dictionary) refers to a data structure that can store `<key, value>` pairs where the keys must be unique. However, there are instances where we wish to store multiple pairs with the same `key` (but unique value). For example, there may be two students named Jon in a class, but their home address may differ. For these situations, we need a *multimap*: a data structure storing `<key, value>` pairs permitting multiple pairs with the same `key`. In this lab you will implement two multimaps and compare their relative efficiencies.

The `MultiMap` Interface

For this lab we will use the following `MultiMap` interface shown below.

```
public interface MultiMap<Key, Value>
{
    public int size();
    public boolean isEmpty();

    /** Returns true if this symbol table contains the specified key. */
    public boolean contains(Key key);

    /** Returns true if this symbol table contains the specified <key, value> pair. */
    public boolean containsPair(Key key, Value value);

    /** Returns all values associated with the specified key in this symbol table.  */
    public Iterable<Value> getAll(Key key);

    /** Inserts the key-value pair into the symbol table (if not already contained). */
    public void put(Key key, Value val);

    /** Removes the specified <key, value> pair value from this symbol table */
    public void delete(Key key, Value value);

    /** Removes all pairs with the given key. */
    public void deleteAll(Key key);

    /** Returns the set of unique keys in the ST as an iterable set */
    public Iterable<Key> keySet();
}
```

What You Need to Do: The `LinkedMultiMap` Implementation

The first symbol table you will implement will be a modification of a singly-linked list: `LinkedMultiMap`. Some of the methods have been implemented for you, others are incomplete. Complete the implementation of `LinkedMultiMap` and verify with the provided junit test class. You are encouraged to add tests.

What You Need to Do: The `MultiHashMap` Implementation

A linked list implementation of a symbol table results in some operations being incredibly efficient, while others are incredibly inefficient. Our goal is to implement a hash table in which collisions are resolved via chaining thus ensuring add, search, and deletion are efficient operations. In our implementation, chains will be implemented as instances of the `LinkedMultiMap`.

One important consideration with the `MultiHashMap` class is that we must expand and contract the table with addition or deletion of pairs. That is, we do not wish for the load of any one chain to be too large thus skewing operation efficiencies. To address this issue, please review the provided code. In particular, the private `resize` method reports when the table will be expanded or contracted. `resize` is then called when needed in the `put`, `delete`, and `deleteAll` methods.

For testing, you should use tests similar to those provided in the `LinkedMultiMapTest` file; we can easily do so since both classes implement the same interface.

There is one important addition that should be included in your testing: appropriate expansion and contraction. All code related to resizing the `MultiHashMap` has been left in place in `MultiHashMap`; do not modify it. You may turn on / off the output produced when resizing by changing the value of `Constants.DEBUG`. A test has been provided in the `MultiHashMapTest` file that adds many `<key, value>` pairs for the table with the intent of observing the size of the table before and after resizing. Sample output is shown here.

```
Resizing from 4(40) to 8
Resizing from 8(80) to 16
Resizing from 16(160) to 32
Resizing from 32(320) to 64
Resizing from 64(640) to 128
Resizing from 128(250) to 64
Resizing from 64(120) to 32
Resizing from 32(60) to 16
Resizing from 16(30) to 8
Resizing from 8(10) to 4
```

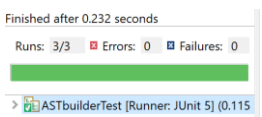
What You Need to Do: Report Timings

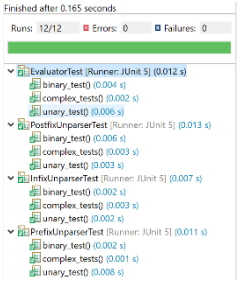
Similar to our last labs, you must report timings of `put` and `deleteAll` operations. Report the results of these timings in the header comment of the respective multimap implementation file. We expect a table and efficiency in *O*-notation similar to what is shown below.

	put	**deleteAll**
500	1	3
1000	2	6
2000	3	9
5000	4	12
…		
	O(?)	O(?)

JUnit Testing

No output should be produced by your tests; we are seeking only a 'green' output indication in Eclipse. Make sure your screenshot shows success of all junit testing methods and not just the overall summary. For example, the image on the left is bad (and on the right is good).





Submitting: Source Code

Your code should be well documented, including docstring comments of methods, blocks of code, and header comments in each file. Junit tests should have reasonable String messages output, if failure occurs.

Header Comments

Your program must use the following standard comment at the top of *each source code file*. Copy and paste this comment and modify it accordingly into these files.

```
/**
 * Write a succinct, meaningful description of the class here. You should avoid wordiness
 * and redundancy. If necessary, additional paragraphs should be preceded by <p>,
 * the html tag for a new paragraph.
 *
 * <p>Bugs: (a list of bugs and / or other problems)
 *
 * @author <your name>
 * @date   <date of completion>
 */
```

Inline Comments

Comment your code with a *reasonable amount of comments* throughout the program. Each method should have a comment that includes information about input, output, overall operation of the function, as well as any limitations that might raise exceptions; Javadoc comments are ideal. Each *block* of code (3-4 or more lines in sequence) in a function should be commented.

It is ***prohibited*** to use *long* comments to the right of lines of source code; attempt 80 character-wide text in source code files.

Submitting: Proof of Program Execution

Execute your code and take a screenshot of the associated junit window and output console (with no output). Place these screenshots into a word processing document (Word, OpenOffice, GoogleDocs, etc.). If multiple screenshots are necessary, label each clearly. Please make sure to crop and enlarge the screenshots so that the picture and / or text is clear (and doesn't strain my old eyes).

Create a PDF of this document and call it `evidence.pdf`.

Source Code Files

Please submit only `LinkedMultiMap`, `MultiHashMap`, and `MultiHashMapTest`.

Final Submission File

In a folder named `lab`, place (1) the source code files and (2) `evidence.pdf`. Zip folder `lab` and label that zip file as `lab.zip`. This zip file is to be submitted via Moodle.

Please be reminded that following instructions explicitly and submitting well-formatted documents (with consistent fonts and typeset equations) is an important part of professionalism.

Only one person in your pair needs to submit the final product.