Implementing Hashtable-based Symbol Tables

Summary: In this assignment, you will implement three versions of a symbol table ADT.

1 Background

In this assignment, you will practice applying your knowledge of ADTs and hashtables, to implement three symbol table ADTs which use relatively standard hashing techniques. Most likely, these will be the most conceptually complicated data types you build. Provided for you in the slides is LinearProbingHashST and SeparateChainingHashST: sample implementations of linear probing and chaining approaches to implementing a hash-based symbol table. Be sure to figure out how they work before attempting the hashtable implementations for this assignment. We will be implementing three techniques for symbol tables:

CompletedTwoProbeChainHT: In a normal chaining approach, keys hash to exactly one index and the key/value pair must reside in the list at that particular index. In this new approach, we will instead calculate two hashes, that indicate two different indices, and then add the new key/value to whichever of two lists, at the two indices, is the shortest. The result is that the chains will end up being shorter since we split in half where the key/value pairs are placed.

CompletedLinearProbingHT: The idea is that we hash to a specific index in the internal array. If the index is empty, we add the key/value pair to it. If occupied, we increment the index by 1 and try again. This repeats until an empty index is found. If the end of the array is reached, then the search will wraparound.

CompletedQuadProbingHT: This technique is very similar to LinearProbingHT, the difference is that it uses a quadratic function to select a target index.

Sample high level UML is shown in Figure 1. Note that your requirement is to follow the interface, not the UML. Hint: following the UML for the two Entry classes WILL save you a headache when initializing the array of generics. Not shown are ProbingHT and TwoProbeChainHT which are interfaces that sit between the SymbolTable interface and the two HT classes (CompletedTwoProbeChainHT, CompletedLinearProbingHT). These extra interfaces are designed to expose some of your hashtable's internals so that we can see what is happening when grading. (The methods are only intended to enable effective grading. In practice, these methods would be omitted from a hashtable ADT since they expose internal state.)

This document is separated into four sections: Background, Requirements, Testing, and Submission. You have almost finished reading the Background section already. In Requirements, we will discuss what is expected of you in this homework. In Testing, we give some basic suggestions on how the hashtable implementations can be tested. Lastly, Submission discusses how your source code should be submitted on Canvas.

2 Requirements [50 points, 6 extra credit]

In this assignment you will implement three types of hash tables. Download the attached Main.java and SymbolTable.java files. The first file defines some tests for the symbol tables, the second is the symbol table interface. Also download ProbingHT.java and TwoProbeChainHT.java which providing testing functionality for the autograder.

- Write a class called CompletedTwoProbeChainHT that implements a two-probe separate chaining hashtable. Two-probe hashing means that you will hash to two positions, and insert the key in the shorter of the two chains. [22 points total]
  - Proper hash calculations:

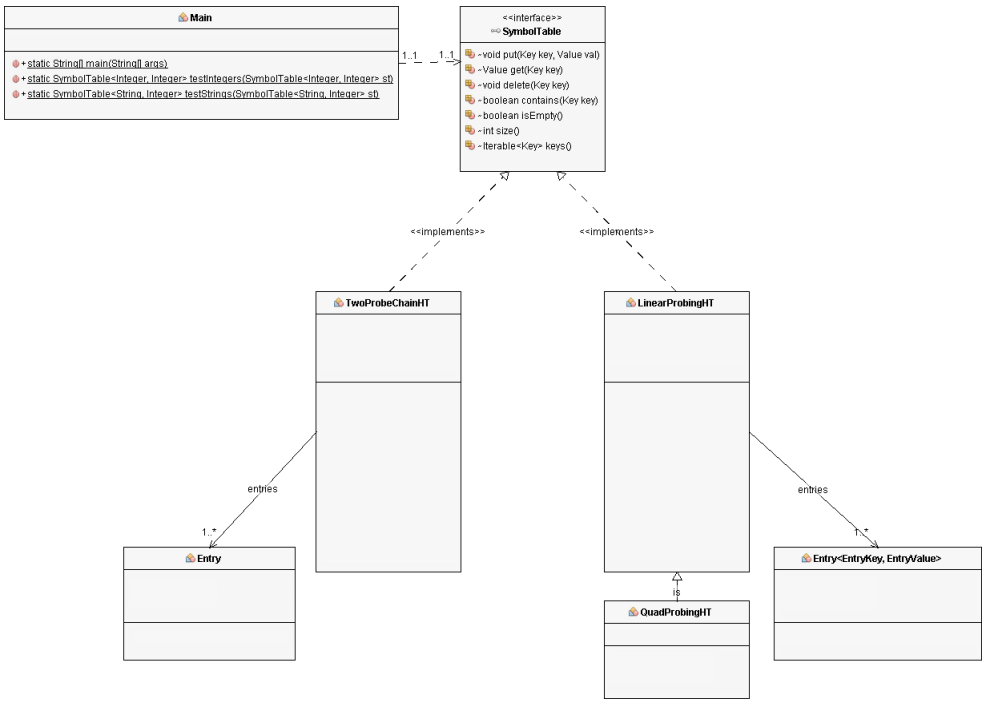

Figure 1: Sample Hashtable Implementation UML.

  - For the first hash function, use the one given in the slides: hash(k)=(k.hashCode() & 0x7fffffff) % M
  - For the second hash function, use: hash2(k)= (((k.hashCode() & 0x7fffffff) % M) * 31) % M
  - Use Java's LinkedList class to store each chain. Do not use SequentialSearchST.
  - **Do not use parallel arrays. (This will be checked manually.)**
- void put(Key key, Value val) - see interface.
- Value get(Key key) - see interface.
- void delete(Key key) - see interface.
- There is no requirement to support array resizing.

- Write a class called CompletedLinearProbingHT that implements a linear probe hashtable. [32 points total = 26 points + 6 extra]
  - Proper hash calculations:
    - Use the basic hash function given in the slides: hash(k, i) = ((k.hashCode() & 0x7fffffff) + i) % M, where k is the key and i is the number of collisions. Each time there is a collision, this will increase the hash by 1.
    - An example hash sequence might look like: 43587, 43588, 43589, 43590, 43581...
    - **Do not use parallel arrays. (This will be checked manually.)**
  - LinearProbingHT() - a constructor that defaults to an array of size 997.
  - void put(Key key, Value val) - see interface.
  - Value get(Key key) - see interface.
  - void delete(Key key) - see interface. Do not degrade performance by using tags or extra nulls; you must update the probe sequence containing the element being deleted. **[6 points extra credit]**
  - boolean contains(Key key) - see interface.

Required Files	Optional Files
CompletedTwoProbeChainHT.java	(none)
CompletedLinearProbingHT.java	
CompletedQuadProbingHT.java	

Table 1: Submission ZIP file contents.

  - boolean isEmpty() - see interface.
  - int size() - see interface.
  - Iterable<Key> keys() - see interface.
  - There is no requirement to support array resizing.

- Write a class called CompletedQuadProbingHT that implements a linear probe hashtable. [2 points]
  - Inherit all the functionality but the hash function from LinearProbingHT.
  - Use the following hash function: hash(k, i) = ((k.hashCode() & 0x7fffffff) + i*i) % M, where k is the key and i is the number of collisions.
  - An example hash sequence might look like: 43587, 43588, 43591, 43596, 43603...

Before submitting to Gradescope, make sure you have implemented CompletedTwoProbeChainHT's getM() and getChainSize(...), and CompletedLinearProbingHT's getM() and getTableEntry(...). Although they are not directly worth points, they are required for the test cases. We have provided sample implementations (see the base files) that you can modify to quickly implement them.

2.1 Packages

Do not import any packages other than java.util.LinkedList, or java.util.Queue. (Do not use any star imports.)

3 Testing

The provided driver file already contains several tests for the operations in the interface. Note that the tests require assertions to be enabled. The tests are designed to check not only the interface operations in isolation, but how they interact with each other. For example, they check that certain properties of the symbol table are invariant over the operations. An invariant is something that does not change. Like the size of the ADT before and after checking if an element is contained. Figure 2 shows the expected output from the driver, once the two classes (TwoProbeChainHT, and LinearProbingHT) have been implemented. Initially, the driver won't compile since it depends on those two classes. *Be aware that while a considerable number of operations are tested, what is provided is not comprehensive.* For example, all tests use the String data type and do not check how well the ADT functions with other types (e.g., Double, a custom class, etc). Please treat these as simple examples of the tests you should creating whenever you write software.

For this assignment, there is no testing write up required. However, you may want to do your own testing to verify the completeness of your implementation.

4 Submission

The submission for this assignment has one part: a source code submission.

Writeup: For this assignment, no write up is required.

Source Code: The source file must be named as shown in the table below, and then added to a ZIP file (which can be called anything). The class must be in the "edu.ser222.m03_04" package, as already done in the provided base file (do not change it!). You will submit the ZIP on Gradescope.

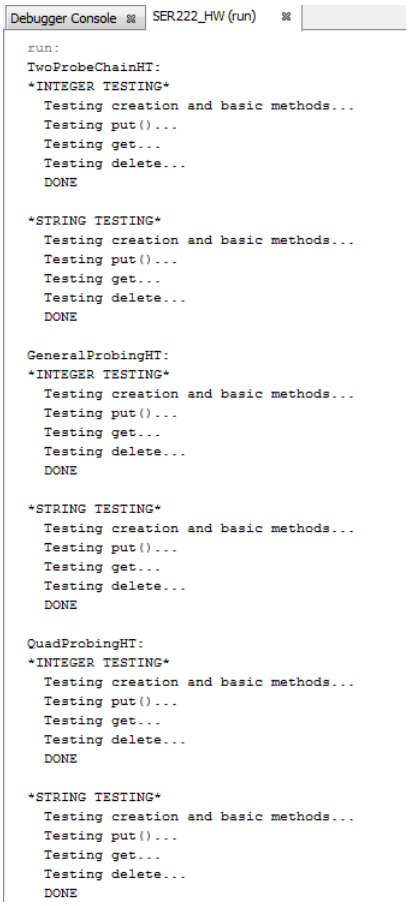

Figure 2: Tree UML Overview

4.1 Gradescope

This assignment will be graded using the Gradescope platform. Gradescope enables cloud-based assessment of your programming assignments. Our implementation of Gradescope works by downloading your assignment to a virtual machine in the cloud, running a suite of test cases, and then computing a tentative grade for your assignment. A few key points:

- **Grades computed after uploading a submission to Gradescope are NOT FINAL.** We have final say over the grade, and may adjust it upwards or downwards.(That said, for this assignment, we don't expect to make any changes.)
- **Additional information on the test cases used for grading is not available, all the information you need (plus some commonsense and attention to detail) is provided in the assignment specification.** Note that each test case will show a small hint in its title about what it is testing that can help you to target what needs to be investigated.

If you have a hard time passing a test case in Gradescope, you should consider if you have made any additional assumptions during development that were not listed in the assignment, and then try to make your submission more general to remove those assumptions.

Protip: the Gradescope tests should be seen as a way to get immediate feedback on your program. This helps you both to make sure you are meeting the assignments requirements, and to check that you are applying your knowledge correctly. Food for thought: if you start on the assignment early, check against our

suite often, and use its feedback, there's no reason why you can't both get full credit and know that you'll get full credit even before the deadline.

4.1.1 Standard Programming Deductions

Due to the nature of cloud grading, we use a different policy for the standard deductions:

- **If your submission does not compile, or is missing the file mentioned above, you will receive a zero grade.**
- **Following the file submission standards (e.g., the submission contains project files, lacks a proper header) is optional, and will not be enforced.** (We would appreciate if you at least included the header though!)