

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:

TwoProbeChainHT: 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.

LinearProbingHT: 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.

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

Sample highlevel 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.)

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 BlackBoard.

2 Requirements [80 points, 8 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.

- Write a class called `TwoProbeChainHT` 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. [34 points total]
 - Proper hash calculations. [5 points]
 - * For the first hash function, use the one given in the slides: $\text{hash}(k) = (k.\text{hashCode()} \& 0x7fffffff) \% M$
 - * For the second hash function, use: $\text{hash2}(k) = (((k.\text{hashCode()} \& 0x7fffffff) \% M) * 31) \% M$
 - * Use Java's `LinkedList` class to store each chain.
 - * Do not use parallel arrays.
 - void `put(Key key, Value val)` - see interface. [10 points]

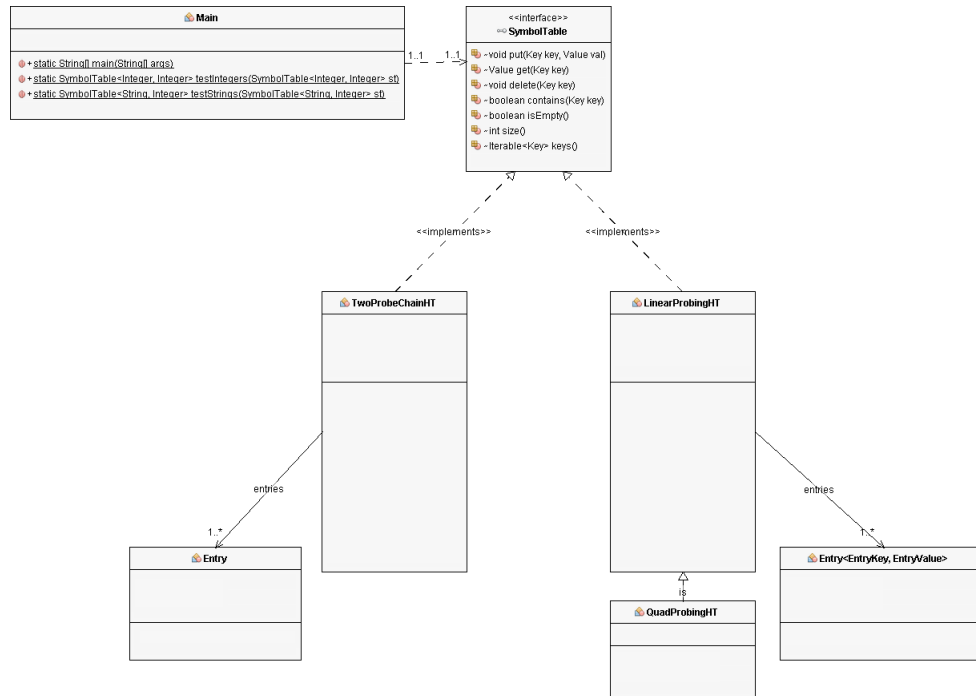


Figure 1: Sample Hashtable Implementation UML

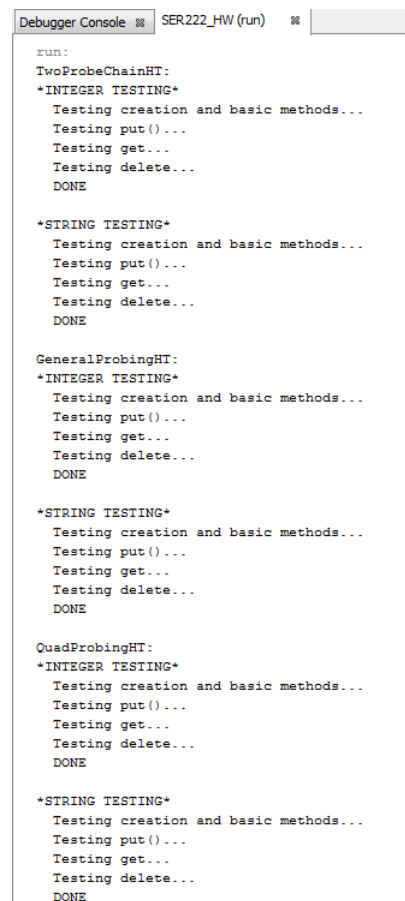
- Value get(Key key) - see interface. [9 points]
- void delete(Key key) - see interface. [10 points]
- Write a class called LinearProbingHT that implements a linear probe hashtable. [38 points total]
 - Proper hash calculations. [4 points]
 - * Use the basic hash function given in the slides: $\text{hash}(k) = (k.\text{hashCode()} \& 0x7fffffff) \% M$, each time there is a collision, increase the hash by 1.
 - An alternative way to structure this hash function is: $\text{hash}(k, i) = ((k.\text{hashCode()} \& 0x7fffffff) + i) \% M$, where k is the key and i is the number of collisions.
 - * An example hash sequence might look like: 43587, 43588, 43589, 43590, 43581...
 - * Do not use parallel arrays.
 - LinearProbingHT() - a constructor that defaults to an array of size 997. [3 points]
 - void put(Key key, Value val) - see interface. [10 points]
 - Value get(Key key) - see interface. [9 points]
 - 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. **[8 points extra credit]**
 - boolean contains(Key key) - see interface. [3 points]
 - boolean isEmpty() - see interface. [3 points]
 - int size() - see interface. [3 points]
 - Iterable<Key> keys() - see interface. [3 points]
 - There is no requirement to support array resizing.
- Write a class called QuadProbingHT that implements a linear probe hashtable. [8 points]
 - Inherit all the functionality but the hash function from LinearProbingHT.

- Use the following hash function: $\text{hash}(k, i) = ((k.\text{hashCode()} \& 0x7\text{ffffff}) + i*i) \% M$, where k is the key and i is the number of collisions.
- An example hash sequence might look like: 43587, 43588, 43591, 43596, 43603...
- All three classes must implement the SymbolTable interface.
- Do not import any packages other than Queue or LinkedList in your hashtable implementations.

3 Testing

The provided driver file already contains several tests for the operations in the interface. Out of the box, the ChainHT implementation should pass these tests. 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., Integer, Double, a custom class, etc).

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.



```

Debugger Console  SER222_HW (run)

run:
TwoProbeChainHT:
*INTEGER TESTING*
  Testing creation and basic methods...
  Testing put()...
  Testing get...
  Testing delete...
  DONE

*STRING TESTING*
  Testing creation and basic methods...
  Testing put()...
  Testing get...
  Testing delete...
  DONE

GeneralProbingHT:
*INTEGER TESTING*
  Testing creation and basic methods...
  Testing put()...
  Testing get...
  Testing delete...
  DONE

*STRING TESTING*
  Testing creation and basic methods...
  Testing put()...
  Testing get...
  Testing delete...
  DONE

QuadProbingHT:
*INTEGER TESTING*
  Testing creation and basic methods...
  Testing put()...
  Testing get...
  Testing delete...
  DONE

*STRING TESTING*
  Testing creation and basic methods...
  Testing put()...
  Testing get...
  Testing delete...
  DONE

```

Figure 2: Tree UML Overview

4 Submission

The submission for this assignment has only one part: a source code submission. It should be uploaded to the submission link on BlackBoard.

Writeup: Not required.

Source Code: Please zip your source code files together as "LastNameST.zip" (e.g. "AcunaST.zip"). It should contain only three files: TwoProbeChainHT.java, LinearProbingHT.java, and QuadProbingHT.java. The classes must be in the default package. Be sure that you use the correct compression format - if you do not use the right format, we may be unable to your submission. Do not include your project files, or leave code in your files that is specific to your IDE/project.