

Lab 17 Hash Table Implementation

Goal

In this lab two different collision resolution schemes will be implemented for a hash table and the resulting performance will be compared with that of linear hashing.

Resources

- Chapter 21: Introducing Hashing
- Chapter 22: Hashing as a Dictionary Implementation
- docs.oracle.com/javase/8/docs/api/ —API documentation for the `java.util.Random` class
- `Lab17Graphs.pdf` – Full size versions of the blank graphs for this lab

In javadoc directory

- `DictionaryInterface.html`—API documentation for the `DictionaryInterface` ADT

Java Files

- `CheckSearchHashTable.java`
- `DictionaryInterface.java`
- `HashPerformance.java`
- `HashedDictionaryOpenAddressingLinear.java`

Introduction

One of the fastest dictionary implementations is the hash table. As long as the table does not become too full, the time for adding and finding an element will be $O(1)$. This performance does not come without a cost. The obvious penalty is that there will be space in the table that is wasted. Another penalty is that the items in the hash table are not in any particular order. Other dictionary implementations will keep items in key order, but it is an inherent property of the hash table that items are not ordered. In fact, as more items are added to the hash table, the size of the table may be increased to maintain the performance. In this case, the items will be rehashed and will no longer be in the same locations or order.

General Collision Resolution

To place an item in a hash table of size m , a hash function $H(k, m)$ is applied to the key k . An integer value between 0 and $m-1$ will be returned and will be the location of the object. If there is already an object in that location, a collision has occurred and must be resolved. In a hash table with open addressing, collisions are resolved by trying other locations until an empty slot is found. One way of viewing this process is that there is a series of hash functions $H_0()$, $H_1()$, $H_2()$, $H_3()$, ... $H_i()$, ..., which are applied one at a time until a free slot is found.

Linear Hashing

For linear hashing, slots in the hash table are examined one after another. From the view of the general scheme, the hash functions are

$$\begin{aligned}H_0(k, m) &= H(k, m) \\H_1(k, m) &= (H(k, m) + 1) \bmod m \\H_2(k, m) &= (H(k, m) + 2) \bmod m \\&\dots \\H_i(k, m) &= (H(k, m) + i) \bmod m\end{aligned}$$

The mod operation is required to keep the values in the range from 0 to $m-1$. While you could use these formulas to compute each of the hash locations, usually the previous value is used to compute the next one.

$$H_i(k, m) = (H_{i-1}(k, m) + 1) \bmod m$$

Linear hashing has the advantage of a simple computational formula that guarantees all the slots will be checked. The performance of linear hashing is affected by the creation of clusters of slots that are filled. Linear hashing tends to create relatively few clusters that are large in size. If there is a collision with a slot inside a cluster, finding an empty slot outside of the cluster will then require a large number of probes. The hash table will have its best performance if the free slots are distributed evenly and large clusters avoided.

Pre-Lab Visualization

Double Hashing

Double hashing is scheme for resolving collisions that uses two hash functions $H(k, m)$ and $h(k, m)$. It is similar to linear hashing except that instead of changing the index by 1, the value of the second hash function is used.

In the view of the general scheme, the hash functions are

$$H_0(k, m) = H(k, m)$$

$$H_1(k, m) = (H(k, m) + h(k, m)) \bmod m$$

$$H_2(k, m) = (H(k, m) + 2 h(k, m)) \bmod m$$

...

$$H_i(k, m) = (H(k, m) + i h(k, m)) \bmod m$$

As with linear hashing, the hash function can be defined in terms of the previous values.

$$H_i(k, m) = (H_{i-1}(k, m) + h(k, m)) \bmod m$$

You must be careful when defining the second hash function.

Suppose that $H(k, m)$ is 12, $h(k, m) = 0$, and $m = 15$. What are the locations that will be probed?



$H_0(k, m)$	$H_1(k, m)$	$H_2(k, m)$	$H_3(k, m)$	$H_4(k, m)$	$H_5(k, m)$

Suppose that $H(k, m)$ is 12, $h(k, m) = 4$, and $m = 15$. What are the locations that will be probed?



$H_0(k, m)$	$H_1(k, m)$	$H_2(k, m)$	$H_3(k, m)$	$H_4(k, m)$	$H_5(k, m)$

If $m=15$, which values of $h(k, m)$ will visit all of the locations in the table?



$h(k, m)$	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Visits all locations?																

Since you really want to probe the entire table, the value returned by the second hash function has some limitations. The first condition is that it should not be 0. The second condition is that it should be relatively prime with respect to m . A common way to guarantee the second condition is to choose a table size that is a prime.

Suppose that you have access to an integer value c that is based on the key
 $c = \text{HashCode}(k)$.

The first hash function will be computed as
 $H(k, m) = c \bmod m$.

Under the assumption that m is a prime, give a formula for computing a second hash function using c . It should return values in the range of 1 to $m-1$.



Double hashing can still be affected by clustering (though to a lesser extent than linear hashing). Every key that has the same value for the second key will probe the table in the same pattern and can still be affected by clusters.

Show how to modify the following code so that it computes the second hash value and then uses it in the search.



```
private int locate(int index, K key)
{
    boolean found = false;

    while ( !found && (hashTable[index] != null) )
    {
        if ( hashTable[index].isIn() &&
            key.equals(hashTable[index].getKey()) )
            found = true; // key found
        else // follow probe sequence
            index = (index + 1) % hashTable.length; // Linear probing
    } // end while

    // Assertion: Either key or null is found at hashTable[index]
    int result = -1;

    if (found)
        result = index;

    return result;
} // end locate
```

Perfect Hashing

In perfect hashing, associated with each key is a unique random sequence of probe locations. Since each key has a unique “view” of the table, the locations of the free slots will be randomly spread out and clustering will be avoided. Even though an approximation to perfect hashing will be implemented in the lab, it is mainly of theoretical interest because perfect hashing is much easier to analyze than linear or double hashing.

Let $s_0(k), s_1(k), s_2(k), \dots$ be a random sequence of values in the range 0 to $m-1$.

$$\begin{aligned} H_0(k, m) &= s_0(k) \\ H_1(k, m) &= s_1(k) \\ H_2(k, m) &= s_2(k) \\ &\dots \\ H_i(k, m) &= s_i(k) \end{aligned}$$

A truly random sequence is not possible, but you can approximate it using pseudo random numbers. To get an idea of how pseudo random numbers can be generated we will look at a once popular algorithm. (Better

algorithms exist and are in use today.) A linear congruential random number generator uses the following simple formula to compute a sequence of numbers:

$$V_{n+1} = (aV_n + c) \bmod m$$

Suppose that $a = 3$, $c = 2$, and $m = 10$, what is the sequence of numbers computed?



V_0	V_1	V_2	V_3	V_4	V_5	V_6	V_7
3							

Is the preceding sequence random? No. It follows a prescribed sequence. If you know one value in the sequence, you know the next value. Further, notice that the preceding sequence misses some of the values between 0 and 9. This means it would not be suitable for a random number generator. In fact, most values for a , c , and m do not result in good pseudorandom number generators, so it is best to create your own random number generator. Instead, use a professionally designed random number generator that has been subjected to a thorough battery of statistical tests.

Even though the sequence is not random, for good choices of values it can appear random which is sufficient for our algorithm. In addition, the fact that the values are actually not random is crucial for the implementation of the perfect hashing algorithm. Each time a value is searched for, you must follow exactly the same sequence. The first value in the sequence is called the seed. If you initialize the pseudorandom number generator with a given seed, it will always produce the same sequence of numbers. This will be the basis for the probe sequence used in perfect hashing.

Show how you can create a new random number generator of the type `Random` from the package `java.util`. Use `c = hashCode(k)` as the seed for the random number generator.



Show how to modify the following code so that instead of receiving a starting index it gets a random number generator. To find the index use the method `nextInt(int k)` method with k as the table size. This will guarantee a random value in the range of 0 to $k-1$.



```
private int locate(int index, K key)
{
    boolean found = false;

    while ( !found && (hashTable[index] != null) )
    {
        if ( hashTable[index].isIn() &&
            key.equals(hashTable[index].getKey()) )
            found = true; // key found
        else // follow probe sequence
            index = (index + 1) % hashTable.length; // Linear probing
    } // end while

    // Assertion: Either key or null is found at hashTable[index]
    int result = -1;

    if (found)
        result = index;

    return result;
} // end locate
```

Creating Random Search Keys

The average time to locate a value in a hash table is usually given based on whether the value is in the table or not. To insert a new value in a table, you must probe for a free slot. Thus the time to insert a new value is basically the same as the time to search for a value that is not in the table. To search for a value that is in the table, you must follow the same pattern of probes that was used when the value was inserted. The average will include values that were inserted early and thus typically require fewer probes to locate.

To test both kinds of searches, an array of unique random words will be created. The first half of the array will be inserted into a hash table. The average of finding a value in the first half of the array will give the average for successful searches and the average over the second half will give the average for unsuccessful (failure) searches.

To make the test more interesting, random three-syllable pseudowords will be created. It is possible that a word will be generated twice. To avoid placing such words in the array, you will have to test to see if the word has been generated before. Using a hash table is a perfect way to do this. As a word is generated, check the hash table to see if it has been generated before. If not, add it to the array and the hash table.

Write an algorithm that creates the array of unique random words. You may assume that three arrays of syllables (`firstSyl`, `secondSyl`, and `thirdSyl`) have already been created. Further, assume that a random number generator of type `Random` from the package `java.util` has been created. It may be helpful to use the method `nextInt(int k)` which will return a random integer from 0 to $k-1$.



The Average Number of Words Generated to Get a Unique Word

One thing to consider is the number of words that will be generated before a unique value is found. As more unique words are generated, it becomes more and more likely that you will randomly generate a previously created word and have to discard it. This may become too much of a burden. Let's find out how much extra work will be required.

To do so, the probability that more than one word will be generated needs to be computed. A probability is a real value between 0 and 1. It tells you the likelihood that an event occurs. An event with a probability of 0.5 has a 50% chance of occurring. An event with a probability of 1 is certain.

Suppose that there are a total of $T = 1000$ unique words that can be created. (In general, T will be the sizes of the three syllable arrays multiplied together.) Suppose further that 600 words have already been generated.

What is the probability that a randomly generated word will be one that has been generated before?



What is the probability that a randomly generated word will not be one that has been generated before?



To determine the average number of words that will need to be generated in order to get a unique word, one must consider all the possible events (number of words generated to get the unique word) along with their probabilities. Multiplying the number of words needed for each event by the corresponding probability and then adding all the products together gives the average. Lets create a table with that information (under the assumption that 600 of 1000 unique words have been found already).

Event	Probability	Value	Product
1 word generated (1 unique)			
2 words generated (1 duplicate, 1 unique)			
3 words generated (2 duplicates, 1 unique)			
4 words generated (3 duplicates, 1 unique)			
5 words generated (4 duplicates, 1 unique)			
6 words generated (5 duplicates, 1 unique)			
7 words generated (6 duplicates, 1 unique)			
8 words generated (7 duplicates, 1 unique)			
9 words generated (8 duplicates, 1 unique)			
...			

The probability that only one word is generated will be just the probability that the first word generated is not one that has been generated before. Write that probability in the table.



The probability that two words are generated will be the product of the probability that the first word was generated before times the probability that the second word was not generated before. Write that probability in the table.



The probability that three words are generated will be the product of the probability that the first word was generated before times the probability that the second word was generated before times the probability that the third word was not generated before. Write that probability in the table.



There is a pattern here. Using that pattern complete the probabilities column in the table.



The values will just be the number of words generated. Fill in the value column in the table.



For each row in the table multiply the probability by the value and record the result in the product column.



Add all the products together and record the sum here.



To get the exact answer, you must add up an infinite number of terms. The product is composed of two parts. One part is getting smaller exponentially and the other is getting larger linearly. Eventually the exponential part will dominate and the sum will converge. For the given situation, nine terms will give an answer that is reasonably close to the exact value of 2.5.

As long as no more than 60% of the possible words have been generated, the number of extra words that get generated to find each unique word will be less than 1.5 and will not require too much extra effort.

The syllable arrays given in the lab each have a size of 15. How many possible words are there?



What is 60% of this total?



To test hash tables with more data values than this, the size of the syllable arrays should be increased.

Counting the Number of Probes

Consider again the code that locates an item or a free slot in the hash table using linear probing.

```
private int locate(int index, K key)
{
    boolean found = false;


    while ( !found && (hashTable[index] != null) )
    {
        if ( hashTable[index].isIn() &&
            key.equals(hashTable[index].getKey()) )
            found = true; // key found
        else // follow probe sequence
            index = (index + 1) % hashTable.length; // Linear probing
    } // end while

    // Assertion: Either key or null is found at hashTable[index]
    int result = -1;

    if (found)
        result = index;

    return result;
} // end locate
```

Suppose that index is 2 and key is 57. Trace the code and circle the index of any location that is accessed.



0	1	2	3	4	5	6	7	8	9
	15	25	13	57	19			16	11


How many locations were circled? (How many probes were made?)



How many times did the body of the loop execute?



Now suppose that index is 2 and key is 99. Trace the code again and circle the index of any location that is accessed.



0	1	2	3	4	5	6	7	8	9
	15	25	13	57	19			16	11

How many locations were circled? (How many probes were made?)



How many times did the body of the loop execute?



It should be the case that only one of the traces had the same number of loop executions as probes.

Show how to modify the `locate()` method so that it will add the number of probes made to a static variable named `totalProbes`.



Directed Lab Work

A hash table class with linear collision resolution has already been implemented in the `HashedDictionaryOpenAddressingLinear` class. You will implement three new classes `HashedDictionaryOpenAddressingLinearInstrumented`, `HashedDictionaryOpenAddressingDoubleInstrumented`, and `HashedDictionaryOpenAddressingPerfectInstrumented` based on that class. They will allow you to gather statistics about the number of probes made to insert values. The `HashPerformance` class will generate random arrays of keys, insert the keys in the various kinds of hash tables, and then display the averages.

Implementing Double Hashing

Step 1. If you have not done so, look at the implementation of a hash table with linear probing in `HashedDictionaryOpenAddressingLinear.java`. Look at the code in `CheckSearchHashTable.java`. Compile the classes `CheckSearchHashTable` and `HashedDictionaryOpenAddressingLinear`. Run the `main()` method in `CheckSearchHashTable`.

Checkpoint: The program will ask you for the number of trials, the number of data values, and a seed. Enter 1, 1000, and 123, respectively. An array of 1000 random values between 0 and 1000 should be generated. The first 500 of those values will be inserted into a hash table. The code will check that searches work correctly. The first 250 values in the array will then be removed from the hash table. Again searches will be checked. Finally, the last 500 values in the array will be added into the hash table. Again searches will be checked.

Verify that the code passed each of the three tests.

The first goal is to create the class for double hashing and verify that it works.

Step 2. Copy `HashedDictionaryOpenAddressingLinear.java` into a new file `HashedDictionaryOpenAddressingDoubleInstrumented.java`.

Step 3. Create a new private method `getSecondHashIndex(Object key)`, which computes a second hash function. Refer to the formula created in the pre-lab exercises.

Step 4. Refer to the pre-lab exercises and modify the `locate()` and `probe()` methods to use double hashing instead of linear hashing.

Step 5. Change the code in `CheckSearchHashTable` so that it creates a new object of type `HashedDictionaryOpenAddressingDoubleInstrumented`. Run the main method in `CheckSearchHashTable`.

Checkpoint: Use 1, 1000, and 123 for the input values.

Verify that the code passed each of the three tests. If not, debug the code and retest.

The next goal is to create the class for perfect hashing and verify that it works.

Implementing Perfect Hashing

Step 6. Copy `HashedDictionaryOpenAddressingLinear.java` into a new file `HashedDictionaryOpenAddressingPerfectInstrumented.java`.

Step 7. Create a new private method `getHashGenerator(Object key)` which will create and return a random number generator that will be used to generate the sequence of probes. Refer to your answer from the pre-lab exercise.

Step 8. Again, refer to the pre-lab exercises and modify the `locate()` and `probe()` methods to use perfect hashing instead of linear hashing. (Remember to change the first argument to be a random number generator instead of an integer.)

Step 9. Find all places where the `locate()` and `probe()` methods are called and change it so that `getHashGenerator` is called instead of `getHashIndex`. Once you are finished, there should no longer be any calls to `getHashIndex`. Remove the `getHashIndex` method.

Step 10. Change the code in `CheckSearchHashTable` so that it creates a new object of type `HashedDictionaryOpenAddressingPerfectInstrumented`. Run the main method in `CheckSearchHashTable`.

Checkpoint: Use 1, 1000, and 123 for the input values.

Verify that the code still passes each of the three tests. If not, debug the code and retest.

Adding Statistics

Step 11. Copy `HashedDictionaryOpenAddressingLinear.java` into a new file `HashedDictionaryOpenAddressingLinearInstrumented.java`.

Step 12. Refer to the pre-lab exercises and add in code to the `locate()` and `probe()` methods that will count the number of probes.

Step 13. Change the code in `CheckSearchHashTable` so that it creates a new object of type `HashedDictionaryOpenAddressingLinearInstrumented`. Run the main method in `CheckSearchHashTable`.

Checkpoint: Use 1, 1000, and 123 for the input values.

Verify that the code still passes each of the three tests. If not, debug the code and retest.

Step 14. Make similar changes in `HashedDictionaryOpenAddressingDoubleInstrumented` and `HashedDictionaryOpenAddressingPerfectInstrumented`.

Step 15. Change the code in `CheckSearchHashTable` so that it creates a new object of type `HashedDictionaryOpenAddressingDoubleInstrumented`. Run the `main()` method in `CheckSearchHashTable`.

Checkpoint: Use 1, 1000, and 123 for the input values.

Verify that the code still passes each of the three tests. If not, debug the code and retest.

Step 16. Change the code in `CheckSearchHashTable` so that it creates a new object of type `HashedDictionaryOpenAddressingPerfectInstrumented`. Run the `main()` method in `CheckSearchHashTable`.

Checkpoint: Use 1, 1000, and 123 for the input values.

Verify that the code still passes each of the three tests. If not, debug the code and retest.

Generating Random Keys

Step 17. Finish the method `generateRandomData()` in the class `HashPerformance`. Refer to the algorithm in the pre-lab exercises.

Step 18. Compile the code in `HashPerformance` and run the `main()` method.

Checkpoint: The code will ask for the number of items to insert, the number of trials, and the maximum load factor for the hash table. Use 1, 10, and 0.9 for the input values.

If all has gone well, the total number of probes will be 10 and the average will be 1.

Checkpoint: The code will ask for the number of items to insert, the number of trials, and the maximum load factor for the hash table. Use 10, 10, and 0.9 for the input values.

If all has gone well, the total number of probes will be approximately 102 and the average will be 1.02. Check that the strings in each array are all different.

Checkpoint: The code will ask for the number of items to insert, the number of trials, and the maximum load factor for the hash table. Use 80, 10, and 0.9 for the input values.

If all has gone well, the total number of probes for linear hashing will be approximately 1900. The total number of probes for double and perfect hashing should be about 1550. In general, perfect hashing is expected to take slightly fewer probes than double hashing, but either one could be better.

Insert Performance

Step 19. Run `HashPerformance` for different numbers of items to be inserted into the hash table and record the results in the following two tables. In each case, enter 10 for the number of trials and 0.5 for the maximum load for the hash table.

Average Number of Probes for the Three Kinds of Hash Tables:

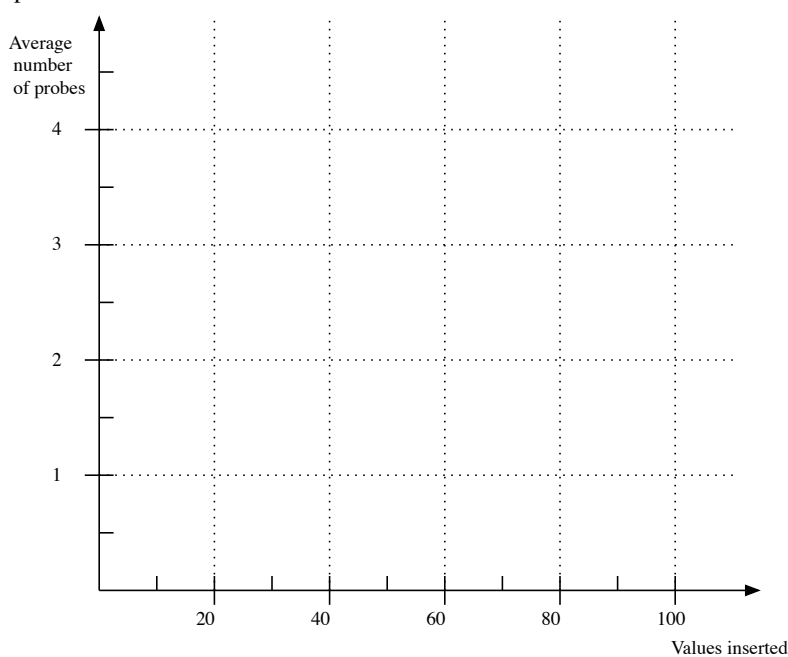
NUMBER OF ITEMS INSERTED	AVERAGE PROBES FOR LINEAR HASHING	AVERAGE PROBES FOR DOUBLE HASHING	AVERAGE PROBES FOR PERFECT HASHING
10			
20			
30			
40			
50			
60			
70			
80			
90			
100			
110			

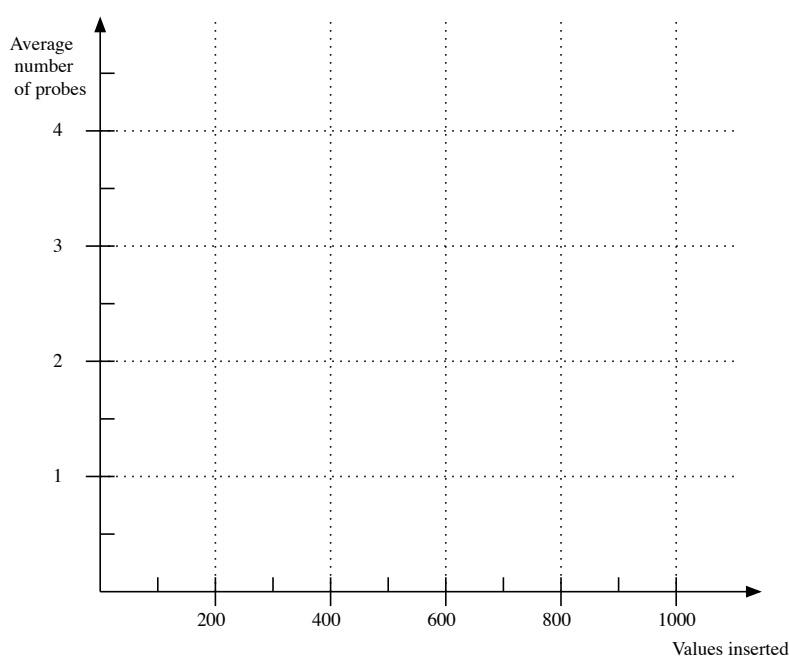
You should notice a sudden jump in the number of probes needed. The hash table resizing itself and then rehashing all the items causes this. The average cost for the insertions after a resize will show a decrease as the cost of the resizing is spread out over the insertions that follow. At approximately what values did a resizing occur?

Average Number of Probes for the Three Kinds of Hash Tables:

NUMBER OF ITEMS INSERTED	AVERAGE PROBES FOR LINEAR HASHING	AVERAGE PROBES FOR DOUBLE HASHING	AVERAGE PROBES FOR PERFECT HASHING
100			
200			
300			
400			
500			
600			
700			
800			
900			
1000			

Step 20. Plot the average number of probes for each of the three kinds of hash tables on the following two graphs.





Insertion Performance versus Initial Table Size

The cost of resizing the table is a hidden cost that gets spread out over all of the insertions. If you can accurately predict the number of data values to be inserted, this hidden cost can be avoided. Just set the initial size of the table to be large enough to accommodate all the data for the given load factor.

Step 21. In the class `HashPerformance`, add code that will query the user for the initial size of the hash table. It should be a positive integer value.

Step 22. Modify the invocation of the constructor for each of the three kinds of hash tables to take as an argument the value that was read in.

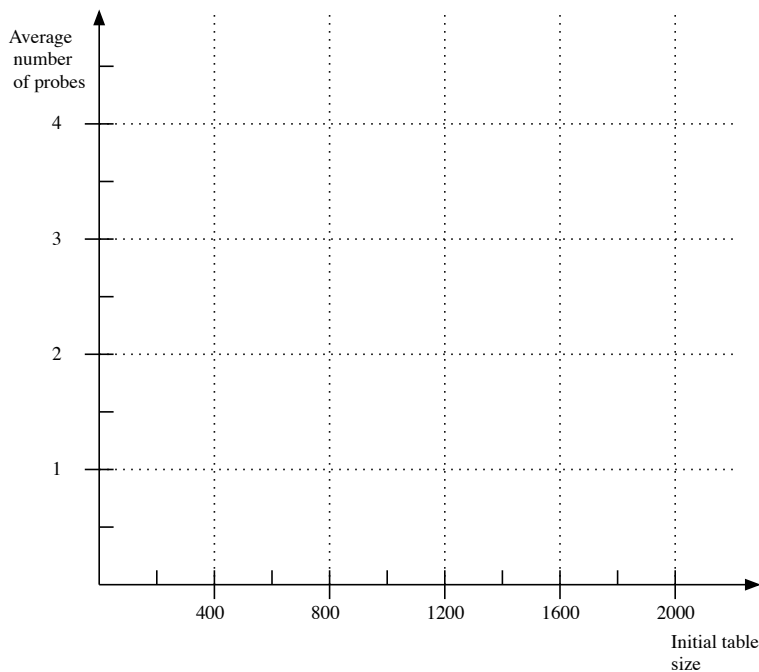
Step 23. Run `HashPerformance` for different initial sizes of the hash table and record the results. In each case, enter 1000 for the number of items to insert, 10 for the number of trials, and 0.5 for the maximum load.

Average Number of Probes with Respect to the Initial Table Size:

INITIAL TABLE SIZE	AVERAGE PROBES FOR LINEAR HASHING	AVERAGE PROBES FOR DOUBLE HASHING	AVERAGE PROBES FOR PERFECT HASHING
50			
100			
250			
500			
1000			
2000			

In each of the cases, the final size of the hash table will be about 2000.

Step 24. Plot the data on the following graph.



Search Performance versus Load Factor

The cost of searching for an item in a hash table is not affected by resizing. Code will be added to distinguish between the number of probes required to search for items in the table (successful search) and items that are not in the table (failure or unsuccessful search).

Step 25. In the class `HashPerformance`, add a method `insertHalfData()` that is based on `insertAllData()`. It will insert just the first half of the array into the hash table.

Step 26. In the class `HashPerformance`, add the methods `searchFirstHalf()` and `searchSecondHalf()` that search for each of the keys in the first and second half of the array, respectively. Use the method `contains()` to determine if the value is in the hash table.

Step 27. Change the call to `generateRandomData()` so that it uses `2*insertCount` instead of `insertCount`.

Step 28. Change the code in the `main()` of `HashPerformance` so that it calls `insertHalfData()` instead of `insertAllData()`.

Step 29. After the code that records the number of probes for the insertions, add code that calls `resetTotalProbes`, performs `searchFirstHalf`, and then finally records the number of probes needed for the successful searches. Do this for each of the three kinds of hash tables.

Step 30. After that code, add code that calls `resetTotalProbes`, performs `searchSecondHalf`, and finally records the number of probes needed for the unsuccessful searches. Do this for each of the three kinds of hash tables.

Step 31. Compile and run `HashPerformance`.

Checkpoint: Enter 100 for the number of values to insert, 1000 for the number of trials, 0.75 for the maximum load factor, and 75 for the initial table size.

The values should be close (typically a difference between -0.1 to 0.1) to the ones listed in the following table. If the values are close but not within the desired range, run the code again with the same values and recheck the results. If the values are still not close, carefully examine the code for errors.

	Average Number of Probes to Insert the Data	Average Number of Probes for Successful Searches	Average Number of Probes for Unsuccessful Searches
Linear Hashing	3.1	1.7	3.6
Double Hashing	2.6	1.5	2.6
Perfect Hashing	2.6	1.5	2.6

Step 32. Compile and run HashPerformance for different load factors. The load factor is the number of data items in the table divided by the table size. The initial table size and maximum load factor will be chosen so that the table does not resize and the desired load factor is achieved once all the values have been inserted into the hash table. In each case, enter 10 for the number of trials, 0.99 for the maximum load for the hash table, and 1000 for the initial table size.

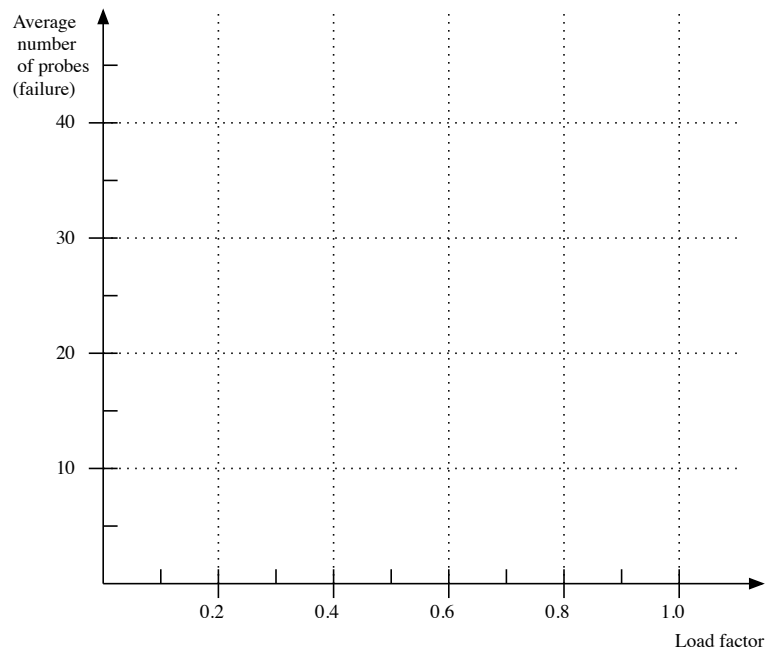
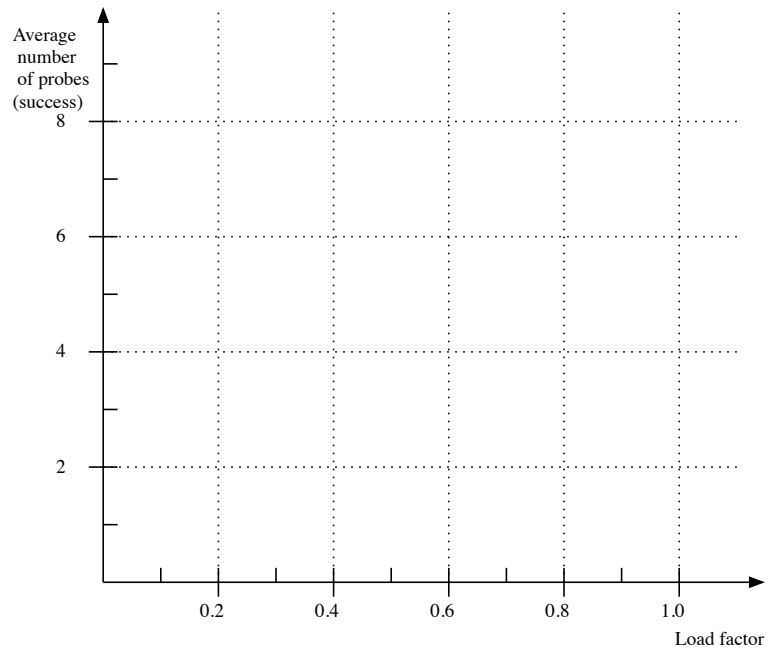
Average Number of Probes for Successful Searches:

LOAD FACTOR (N = NUMBER OF DATA VALUES TO INSERT)	AVERAGE PROBES FOR LINEAR HASHING	AVERAGE PROBES FOR DOUBLE HASHING	AVERAGE PROBES FOR PERFECT HASHING
0.30 (n=300)			
0.40 (n=400)			
0.50 (n=500)			
0.60 (n=600)			
0.70 (n=700)			
0.75 (n=750)			
0.80 (n=800)			
0.85 (n=850)			
0.90 (n=900)			
0.95 (n=950)			

Average Number of Probes for Failed Searches:

LOAD FACTOR	AVERAGE PROBES FOR LINEAR HASHING	AVERAGE PROBES FOR DOUBLE HASHING	AVERAGE PROBES FOR PERFECT HASHING
0.30 (n=300)			
0.40 (n=400)			
0.50 (n=500)			
0.60 (n=600)			
0.70 (n=700)			
0.75 (n=750)			
0.80 (n=800)			
0.85 (n=850)			
0.90 (n=900)			
0.95 (n=950)			

Step 33. Plot the data on the following graphs.



Post-Lab Follow-Ups

1. Add two methods `startSearch()` and `endSearch()` to the instrumented hash table classes. The method `startSearch()` will be called at the beginning of `locate()` and `probe()`. The method `endSearch()` will be called at the end of `locate()` and `probe()`. Use these methods to compute the maximum and minimum numbers of probes needed by `locate()` and `probe()`.
2. Modify the methods added in the pervious exercise to compute the standard deviation of the number of probes needed. See the followup questions to Lab 8 on sorting for more on computing a standard deviation.
3. Change the `enlargeHashTable()` method so that instead of doubling in size it increases the size by a constant value (set it to the initial size of the hash table.) Recalculate the insertion performance of the three kinds of hash tables and plot.
4. Change the `enlargeHashTable()` method so that it increases the size of the hash table by a constant factor (set it to 20%). Recalculate the insertion performance of the three kinds of hash tables and plot.
5. Create a new class that implements quadratic probing. Add code to count the number of probes. Compute the performance and plot.
6. Create code that will test the performance of a hash table when there are insertions and deletions mixed together. Start by adding N values to the table. Then do a sequence of K operations. For each operation, randomly choose to add or remove a value with a probability of 0.5. Print the number of probes needed for the K operations. Once the K operations are finished, compute the performance in terms of the number of probes for successful and unsuccessful searches. Plot the results for $K = 100, 1000, \text{ and } 10000$ with values of N that are 200, 400, 600, 800, 1000.
7. As the number of entries that are marked as removed increases, the performance of failed searches will degrade. Add code that keeps track of the slots that are empty but marked as removed. Compute an effective load factor that is based on the number of filled slots plus the number of slots marked as removed. If this value ever gets larger than the maximum load factor create a new table and rehash. The size of the new table can depend on the actual load factor based on the number of entries. If the actual load factor is small enough (less than $\frac{1}{4}$ the maximum load factor) create a smaller table. If the actual load factor is large enough (greater than $\frac{3}{4}$ the maximum load factor) create a larger table. Otherwise, keep the size the same.

Insert N items into the table initially. Compute the performance with and without this rehashing when adding and removing K blocks of M randomly chosen items.
8. Each random word that was created in the lab can be represented as a triplet of three numbers. Create a method that given a triplet will return the next triplet in a lexicographic order. Given a random triplet as a starting point, use this method to generate N words without any repetition. To make the sequence of words look more random, you can skip over K triplets for each word generated. As long as K is relatively prime with respect to the total number of possible triplets, every triplet will be visited before there is a repetition. (This is similar to the search pattern that double hashing does.)
9. If perfect hashing into a table with load factor $\alpha = N / M$ has a probability of $1 - \alpha$ of finding an empty slot, compute the average number of probes needed to find an empty slot. This computation is similar to the one done in the Pre-Lab to determine the average number of random words you must generate to get a unique word when duplicates are to be discarded. This computation can be done exactly.

