**Hashing**

What is the absolute fastest way to insert and retrieve data? We want O(1) efficiency. What if the data could be used to calculate its own index in the array, both for inserting and retrieving? *Hashing* is the name for that technique. The simplest kind of *hash table* is an array. A *hash function* calculates the storage index in the array from the item itself. (Hashing is also used to detect file tampering, to keep passwords secure, for digital signatures, for matching algorithms, and for Bitcoin’s proof-of-work.)

As a simple example, suppose a company stores customer orders in an array according to the last two digits of the customer’s phone number, using the *hash function* phone\_number % 100. Thus two customers whose phone numbers are 257-3178 and 253-5169 respectively will have their orders stored in list[78] and list[69]. Unfortunately, two phone numbers may hash to the same index, resulting in a *collision*. In this lab, you will deal with such collisions in three different ways, described below.

A good hash function minimizes such collisions by spreading the hash addresses uniformly around the table. Every object in Java has a hashCode method (see the cheat sheet) which hashes on the object’s hexadecimal address. A common hash function is Math.abs(obj.hashCode() % array.length), which is what we use in our program. The shell has three methods for you to complete, one for each way to deal with collisions.

1. **Linear Probing**

If there is a collision, simply try the next index in the array. If it is null, then store the item. If it is not, then try the next index, and so on. If you reach the end of the array, treat it as a circular structure and go back to the beginning of the array to search for an empty spot.

1. **Rehashing**

If there is a collision, then compute a new index using the old hash address as input. Repeat if necessary. A common rehash function has the form (hash\_address + k) % array.length where k and array.length are relatively prime, which ensures that every index will be covered. If you choose a k that is not relatively prime to the length of the array, you run the risk of an infinite loop.

For example, the hash function for this table is (datum % 10). The rehash function is (hash\_address + 3) % 10. Note that 3 and 10 are relatively prime. Here are the steps to insert 26402 into the table:

|  |  |
| --- | --- |
| [0] |  |
| [1] | 27401 |
| [2] | 68902 |
| [3] |  |
| [4] |  |
| [5] | 67905 |
| [6] |  |
| [7] |  |
| [8] |  |
| [9] | 27309 |

26402 % 10 🡪 2 (taken)

(2+3) % 10 🡪5 (taken)

(5+3) % 10 🡪8, which becomes the hash address of the new item.

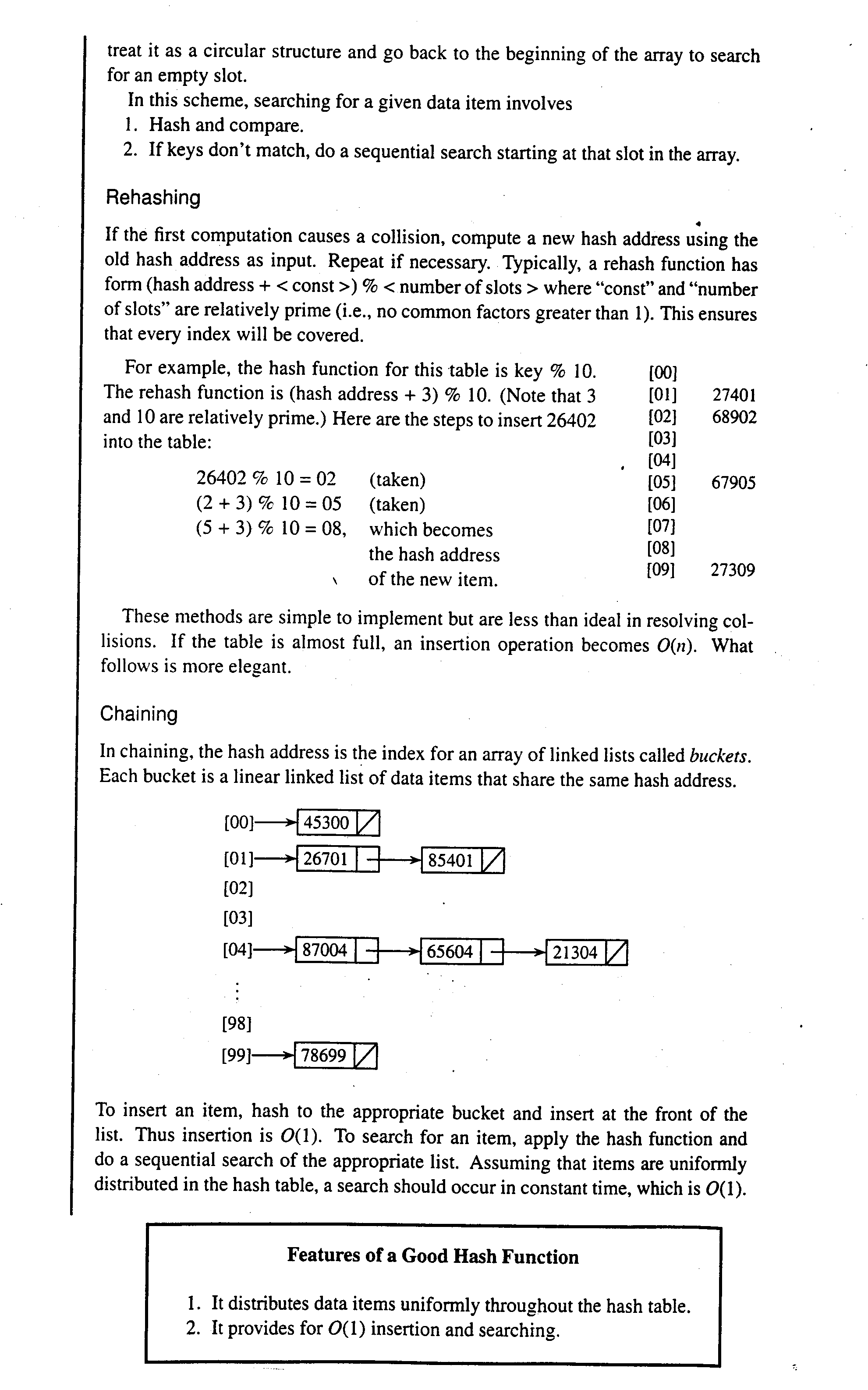
You will need to generate a k which is relatively prime with any given array.length. To help you think about the algorithm:

if length of the array is 10, then the first relatively prime number is \_3\_\_.

if length of the array is 15, then the first relatively prime number is \_2\_\_.

if length of the array is 6, then the first relatively prime number is \_\_5\_.

1. **Chaining**



When dealing with collisions by chaining, the hash function returns the index in an array of linked lists called *buckets*. Each bucket is a linked list of data items that share the same hash address. Our lab uses java.util.LinkedList, but this picture shows standard ListNodes.

To insert an item, hash to the bucket and insert at the front of the list. Thus insertion is O(1). To search for an item, apply the hash function and do a sequential search of the linked list. If the array is large enough, and the hash function distributes the hash addresses uniformly, then the number of collisions will be small, and searching (e.g. contains(), indexOf() ) will occur in an amortized O(1).

**Hashing Class Design**

One aspect of this lab is to understand hashing. What is hashing? Using data to find its own index.

Another aspect of this lab is to understand the design decisions that the teacher made. In this lab, it seems that we are to do the "same" things, but in three different ways (linear probing, rehashing, and chaining). Therefore, it seems like we need a superclass and three subclasses.

Do the three subclasses share any data and/or data structures? If so, then the superclass should be an abstract class. If not, then the superclass should be an interface. Since two of our three subclasses have the same data structure (an array of objects), but the third does not (it has an array of LinkedLists), it seemed better to the teacher to make the superclass an interface. Here it is:

interface Hashtable  
 {  
 void add(Object obj);  
 boolean contains(Object obj);

Object[] getArray(); // for Codepost  
 }

Explain what an interface guarantees. Abstract methods WILL BE IMPLEMENTED

Explain what is happening on lines 28, 32, 35, and 38. Superclass references to a subclass object

Which lines allow for polymorphic behavior? \_\_\_\_\_\_\_\_\_ and \_\_\_\_\_\_\_\_\_.

Which methods will show polymorphic behavior? \_\_\_add\_\_\_\_ and \_\_contains\_\_.

22 int scheme = Integer.parseInt(JOptionPane.showInputDialog(  
 23 "The Load Factor is " + (double)numItems/arrayLength +  
 24 "\nWhich collision scheme?\n"+  
 25 "1. Linear Probing\n" +  
 26 "2. Rehashing\n"+  
 27 "3. Chaining"));  
 28 Hashtable table = null;  
 29 switch( scheme )  
 30 {  
 31 case 1:   
 32 table = new HashtableLinearProbe(arrayLength);  
 33 break;  
 34 case 2:   
 35 table = new HashtableRehash(arrayLength);  
 36 break;  
 37 case 3:   
 38 table = new HashtableChaining(arrayLength);  
 39 break;  
 40 default: System.exit(0);   
 41 }  
 42 for(int i = 0; i < numItems; i++)  
 43 table.add("Item" + i);  
 44  
 45 int itemNumber = -1;  
 46 while(true)  
 47 {  
 48 itemNumber = Integer.parseInt(JOptionPane.showInputDialog(  
 49 "Search for: Item0" + " to "+ "Item"+(numItems-1)));  
 50 if(itemNumber == -1)  
 51 System.exit(0);  
 52 System.out.println("Searching for Item"+ itemNumber);  
 53 String key = "Item" + itemNumber;  
 54 boolean found = table.contains(key);   
 55 }   
 56 }