TIP

Important: Always make the array size a prime number. Use 59 instead of 60, for example. (Other primes less than 60 are 53, 47, 43, 41, 37, 31, 29, 23, 19, 17, 13, 11, 7, 5, 3, and 2.) If the array size is not prime, an endless sequence of steps may occur during a probe. If this happens during a Fill operation, the applet will be paralyzed.

The Problem with Quadratic Probes

Quadratic probes eliminate the clustering problem we saw with the linear probe, which is called *primary clustering*. However, quadratic probes suffer from a different and more subtle clustering problem. This occurs because all the keys that hash to a particular cell follow the same sequence in trying to find a vacant space.

Let's say 184, 302, 420, and 544 all hash to 7 and are inserted in this order. Then 302 will require a one-step probe, 420 will require a four-step probe, and 544 will require a nine-step probe. Each additional item with a key that hashes to 7 will require a longer probe. This phenomenon is called *secondary clustering*.

Secondary clustering is not a serious problem, but quadratic probing is not often used because there's a slightly better solution.

Double Hashing

To eliminate secondary clustering as well as primary clustering, we can use another approach: *double hashing*. Secondary clustering occurs because the algorithm that generates the sequence of steps in the quadratic probe always generates the same steps: 1, 4, 9, 16, and so on.

What we need is a way to generate probe sequences that depend on the key instead of being the same for every key. Then numbers with different keys that hash to the same index will use different probe sequences.

The solution is to hash the key a second time, using a different hash function, and use the result as the step size. For a given key the step size remains constant throughout a probe, but it's different for different keys.

Experience has shown that this secondary hash function must have certain characteristics:

- It must not be the same as the primary hash function.
- It must never output a 0 (otherwise, there would be no step; every probe would land on the same cell, and the algorithm would go into an endless loop).

Experts have discovered that functions of the following form work well:

```
stepSize = constant - (key % constant);
```

where `constant` is prime and smaller than the array size. For example,

```
stepSize = 5 - (key % 5);
```

This is the secondary hash function used in the HashDouble Workshop applet. Different keys may hash to the same index, but they will (most likely) generate different step sizes. With this hash function the step sizes are all in the range 1 to 5. This is shown in Figure 11.9.

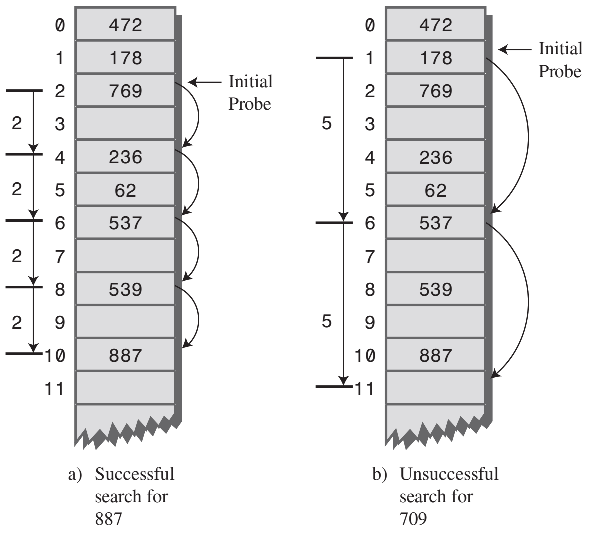

FIGURE 11.9 Double hashing.

The HashDouble Applet with Double Hashing

You can use the HashDouble Workshop applet to see how double hashing works. It starts up automatically in Double-hashing mode, but if it's in Quadratic mode, you can switch to Double by creating a new table with the New button and clicking the Double button when prompted. To best see probes at work, you'll need to fill the table rather full, say to about nine/tenths capacity or more. Even with such high load factors, most data items will be found immediately by the first hash function; only a few will require extended probe sequences.

Try finding existing keys. When one needs a probe sequence, you'll see how all the steps are the same size for a given key, but that the step size is different—between 1 and 5—for different keys.

Java Code for Double Hashing

Listing 11.2 shows the complete listing for `hashDouble.java`, which uses double hashing. It's similar to the `hash.java` program (Listing 11.1), but uses two hash functions: one for finding the initial index and the second for generating the step size. As before, the user can show the table contents, insert an item, delete an item, and find an item.

LISTING 11.2 The `hashDouble.java` Program

```
// hashDouble.java
// demonstrates hash table with double hashing
// to run this program: C:>java HashDoubleApp
import java.io.*;
////////////////////////////////////////////////////////////////
class DataItem
   {                                 // (could have more items)
   private int iData;                // data item (key)
//--------------------------------------------------------------
   public DataItem(int ii)           // constructor
      { iData = ii; }
//--------------------------------------------------------------
   public int getKey()
      { return iData; }
//--------------------------------------------------------------
   }  // end class DataItem
////////////////////////////////////////////////////////////////
class HashTable
   {
   private DataItem[] hashArray;     // array is the hash table
   private int arraySize;
   private DataItem nonItem;         // for deleted items
// -------------------------------------------------------------
   HashTable(int size)               // constructor
      {
      arraySize = size;
      hashArray = new DataItem[arraySize];
      nonItem = new DataItem(-1);
      }
// -------------------------------------------------------------
   public void displayTable()
      {
      System.out.print("Table: ");
      for(int j=0; j<arraySize; j++)
```

LISTING 11.2 Continued

```
         {
         if(hashArray[j] != null)
            System.out.print(hashArray[j].getKey()+ " ");
         else
            System.out.print("** ");
         }
      System.out.println("");
      }
// -------------------------------------------------------------
   public int hashFunc1(int key)
      {
      return key % arraySize;
      }
// -------------------------------------------------------------
   public int hashFunc2(int key)
      {
      // non-zero, less than array size, different from hF1
      // array size must be relatively prime to 5, 4, 3, and 2
      return 5 - key % 5;
      }
// -------------------------------------------------------------
                                     // insert a DataItem
   public void insert(int key, DataItem item)
   // (assumes table not full)
      {
      int hashVal = hashFunc1(key);  // hash the key
      int stepSize = hashFunc2(key); // get step size
                                     // until empty cell or -1
      while(hashArray[hashVal] != null &&
                      hashArray[hashVal].getKey() != -1)
         {
         hashVal += stepSize;        // add the step
         hashVal %= arraySize;       // for wraparound
         }
      hashArray[hashVal] = item;     // insert item
      }  // end insert()
// -------------------------------------------------------------
   public DataItem delete(int key)   // delete a DataItem
      {
      int hashVal = hashFunc1(key);      // hash the key
      int stepSize = hashFunc2(key);     // get step size
```

LISTING 11.2 Continued

```
      while(hashArray[hashVal] != null)  // until empty cell,
         {                               // is correct hashVal?
         if(hashArray[hashVal].getKey() == key)
            {
            DataItem temp = hashArray[hashVal]; // save item
            hashArray[hashVal] = nonItem;       // delete item
            return temp;                        // return item
            }
         hashVal += stepSize;            // add the step
         hashVal %= arraySize;           // for wraparound
         }
      return null;                   // can't find item
      }  // end delete()
// -------------------------------------------------------------
   public DataItem find(int key)     // find item with key
   // (assumes table not full)
      {
      int hashVal = hashFunc1(key);      // hash the key
      int stepSize = hashFunc2(key);     // get step size

      while(hashArray[hashVal] != null)  // until empty cell,
         {                               // is correct hashVal?
         if(hashArray[hashVal].getKey() == key)
            return hashArray[hashVal];   // yes, return item
         hashVal += stepSize;            // add the step
         hashVal %= arraySize;           // for wraparound
         }
      return null;                   // can't find item
      }
// -------------------------------------------------------------
   }  // end class HashTable
////////////////////////////////////////////////////////////////
class HashDoubleApp
   {
   public static void main(String[] args) throws IOException
      {
      int aKey;
      DataItem aDataItem;
      int size, n;
                                  // get sizes
      System.out.print("Enter size of hash table: ");
```

LISTING 11.2 Continued

```
      size = getInt();
      System.out.print("Enter initial number of items: ");
      n = getInt();
                                  // make table
      HashTable theHashTable = new HashTable(size);

      for(int j=0; j<n; j++)      // insert data
         {
         aKey = (int)(java.lang.Math.random() * 2 * size);
         aDataItem = new DataItem(aKey);
         theHashTable.insert(aKey, aDataItem);
         }

      while(true)                 // interact with user
         {
         System.out.print("Enter first letter of ");
         System.out.print("show, insert, delete, or find: ");
         char choice = getChar();
         switch(choice)
            {
            case 's':
               theHashTable.displayTable();
               break;
            case 'i':
               System.out.print("Enter key value to insert: ");
               aKey = getInt();
               aDataItem = new DataItem(aKey);
               theHashTable.insert(aKey, aDataItem);
               break;
            case 'd':
               System.out.print("Enter key value to delete: ");
               aKey = getInt();
               theHashTable.delete(aKey);
               break;
            case 'f':
               System.out.print("Enter key value to find: ");
               aKey = getInt();
               aDataItem = theHashTable.find(aKey);
               if(aDataItem != null)
                  System.out.println("Found " + aKey);
               else
```

LISTING 11.2 Continued

```
                  System.out.println("Could not find " + aKey);
               break;
            default:
               System.out.print("Invalid entry\n");
            }  // end switch
         }  // end while
      }  // end main()
//--------------------------------------------------------------
   public static String getString() throws IOException
      {
      InputStreamReader isr = new InputStreamReader(System.in);
      BufferedReader br = new BufferedReader(isr);
      String s = br.readLine();
      return s;
      }
//--------------------------------------------------------------
   public static char getChar() throws IOException
      {
      String s = getString();
      return s.charAt(0);
      }
//-------------------------------------------------------------
   public static int getInt() throws IOException
      {
      String s = getString();
      return Integer.parseInt(s);
      }
//--------------------------------------------------------------
   }  // end class HashDoubleApp
////////////////////////////////////////////////////////////////
```

Output and operation of this program are similar to those of `hash.java`. Table 11.1 shows what happens when 21 items are inserted into a 23-cell hash table using double hashing. The step sizes run from 1 to 5.

TABLE 11.1 Filling a 23-Cell Table Using Double Hashing

Item		**Hash**		
Number	**Key**	**Value**	**Step Size**	**Cells in Probe Sequence**
1	1	1	4	
2	38	15	2	
3	37	14	3	
4	16	16	4	

TABLE 11.1 Continued

Item Number	**Key**	**Hash Value**	**Step Size**	**Cells in Probe Sequence**
5	20	20	5	
6	3	3	2	
7	11	11	4	
8	24	1	1	2
9	5	5	5	
10	16	16	4	20 1 5 9
11	10	10	5	
12	31	8	4	
13	18	18	2	
14	12	12	3	
15	30	7	5	
16	1	1	4	5 9 13
17	19	19	1	
18	36	13	4	17
19	41	18	4	22
20	15	15	5	20 2 7 12 17 22 4
21	25	2	5	7 12 17 22 4 9 14 19 1 6

The first 15 keys mostly hash to a vacant cell (the 10th one is an anomaly). After that, as the array gets more full, the probe sequences become quite long. Here's the resulting array of keys:

```
** 1 24 3 15 5 25 30 31 16 10 11 12 1 37 38 16 36 18 19 20 ** 41
```

Table Size a Prime Number

Double hashing requires that the size of the hash table is a prime number. To see why, imagine a situation in which the table size is not a prime number. For example, suppose the array size is 15 (indices from 0 to 14), and that a particular key hashes to an initial index of 0 and a step size of 5. The probe sequence will be 0, 5, 10, 0, 5, 10, and so on, repeating endlessly. Only these three cells are ever examined, so the algorithm will never find the empty cells that might be waiting at 1, 2, 3, and so on. The algorithm will crash and burn.

If the array size were 13, which is prime, the probe sequence will eventually visit every cell. It's 0, 5, 10, 2, 7, 12, 4, 9, 1, 6, 11, 3, and so on and on. If there is even one empty cell, the probe will find it. Using a prime number as the array size makes it impossible for any number to divide it evenly, so the probe sequence will eventually check every cell.