Since our mapping from element value to preferred index now has a bit of complexity to it, we might turn it into a method that accepts the element value as a parameter and returns the right index for that value. Such a method is referred to as a *hash function*, and an array that uses such a function to govern insertion and deletion of its elements is called a *hash table*. The individual indexes in the hash table are also sometimes informally called *buckets*.

Our hash function so far is the following:

```
private int hashFunction(int value) {
    return Math.abs(value) % elementData.length;
}
```

Hash Function

A method for rapidly mapping between element values and preferred array indexes at which to store those values.

Hash Table

An array that stores its elements in indexes produced by a hash function.

Collisions

There is still a problem with our current hash table. Because our hash function wraps values to fit in the array bounds, it is now possible that two values could have the same preferred index. For example, if we try to insert `45` into the hash table, it maps to index 5, conflicting with the existing value `5`. This is called a *collision*. Our implementation is incomplete until we have a way of dealing with collisions. If the client tells the set to insert `45`, the value `45` must be added to the set somewhere; it's up to us to decide where to put it.

Collision

When two or more element values in a hash table produce the same result from its hash function, indicating that they both prefer to be stored in the same index of the table.

One common way of resolving collisions is called *probing*, which involves looking for another index to use if the preferred index is taken. For example, if the client wants to add `45` and index 5 is in use, we could just put `45` at the next available index, which is 6 in our example. (Looking forward one index at a time for the next free index is called *linear probing*, but there are other kinds of probing such as *quadratic probing*, which involves jumping around to various places in the hash table.)

Figure 18.5 shows the hash table's state if the values `45`, `91`, and `71` are added and linear probing is used to resolve the collisions. The `45` conflicts with `5` and is put into index 6. The `91` conflicts with existing value `1` and is put into index 2. The `71` conflicts with `1` and `91` and must be put into index 3.

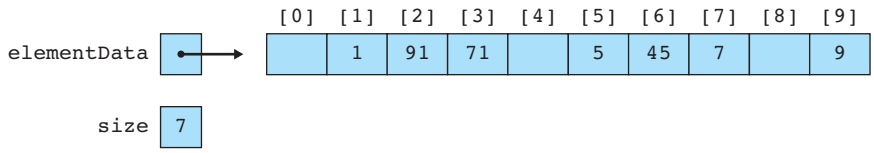

Figure 18.5 Hash collisions resolved by linear probing

Probing gets around the collision problem, but it introduces new problems of its own. For one, it is no longer as simple to find an element's value. A value whose hash function evaluates to *k* might be stored at index *k*, but if something other than *k* is there, it might be at index *k*+1, *k*+2, etc. So we would need to patch our other methods such as `contains` accordingly. We can make the appropriate modifications, but we also have to think about the original goals of this implementation. Searching for elements is supposed to be fast, and if we have to probe through a lot of elements to find anything, we're losing the efficiency we sought after in the first place.

Probing

Resolving hash collisions by placing elements at other indexes in the table rather than their preferred indexes.

Another problem is that the hash table can get full, resulting in no free slots to store a value. The example array of size 10 can hold only 10 elements. If the client tries to add an eleventh element, the array must be resized. Resizing a hash table is a nontrivial operation that we'll discuss in the next section.

Even if the array is not entirely full, if many elements are next to each other it can slow down the runtime for later operations. For example, to add the value `25` to the hash table in Figure 18.5 requires looking at four buckets: indexes 5, 6, 7, and 8 (where the value is finally placed). After doing so, a search for the value `95` (which is not found in the table) would need to examine indexes 5, 6, 7, 8, 9, and 0 before finally giving up. When elements clump up near each other like this, it is called *clustering*, and it is desirable to have as little clustering as possible in a hash table for it to perform efficiently.

Another way of dealing with the collision problem is to change our internal data structure. Collisions would not be a problem if each array index could store more than one value. This is possible if we change the array to be an array of lists rather than an array of integers. The list at index *k* will store all elements that the hash function maps to *k*. Figure 18.6 demonstrates this idea. To add an element to the hash table, we go to its preferred index and add the element value to the list stored at that index. Resolving collisions by storing hash elements in lists is called *separate chaining*. The real `HashSet` and `HashMap` provided in `java.util` use separate chaining internally to resolve collisions.

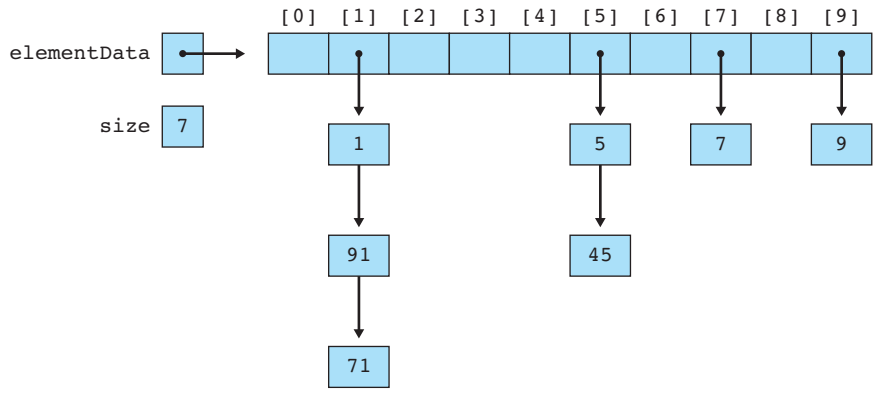

Figure 18.6 Hash collisions resolved by separate chaining

Separate Chaining

Resolving hash collisions by having each index of the table store a list of values rather than a single value.

As with probing, separate chaining faces an efficiency issue if the size of the set becomes large enough, especially if the numbers happen to have a lot of collisions with each other. For example, if the client adds a lot of numbers to the set that end with 1, there will be a long list of elements at index 1 in the hash table, and we'll have to perform a long search through the list to find values. One way to mitigate this issue is to use a larger array to store the element data, which will result in fewer collisions and shorter lists at each index. Another good idea is to use an array size that is not a multiple of 10 or any other round number, to avoid number patterns that are likely to collide. Many hash table implementations use a prime number for the table capacity.

We'll implement the lists of elements as lists of node objects, similar to the implementation of linked lists in Chapter 16. An inner class `HashEntry` represents a single node in such a list:

```
// Represents a single value in a chain stored in one hash bucket.
private class HashEntry {
    private int data;
    private HashEntry next;

    public HashEntry(int data) {
        this(data, null);
    }
```

```
    public HashEntry(int data, HashEntry next) {
        this.data = data;
        this.next = next;
    }
}
```

Using this inner class we can now implement the `add` and `contains` operations on our set. The `add` method adds a new entry to the table if the element is not already found in the table. Before adding an element to the set, we make sure it is not a duplicate by calling `contains`. The `contains` method looks through the list at the appropriate hash bucket index to see if that value is found in the list. When elements are added, we insert them at the front of the appropriate linked list. This is faster than traversing the links all the way to the end of the list to insert the element.

```
// Adds the given element to this set, if it was not
// already contained in the set.
public void add(int value) {
    if (!contains(value)) {
        // insert new value at front of list
        int bucket = hashFunction(value);
        elementData[bucket] = new HashEntry(value, elementData[bucket]);
        size++;
    }
}

// Returns true if the given value is found in this set.
public boolean contains(int value) {
    int bucket = hashFunction(value);
    HashEntry current = elementData[bucket];
    while (current != null) {
        if (current.data == value) {
            return true;
        }
        current = current.next;
    }
    return false;
}
```

We can also implement the `remove` method now. Removal is a bit trickier than adding, because we have to make sure to handle all of the possible cases. If the element to be removed is at the front of its linked list, we must adjust the front reference; otherwise we must change the `next` reference of some existing node in the list. As always with linked lists, we must be careful to check for `null` and not try to traverse past the end of a list.

```
// Removes the given value if it is contained in the set.
public void remove(int value) {
    int bucket = hashFunction(value);
    if (elementData[bucket] != null) {
        // check front of list
        if (elementData[bucket].data == value) {
            elementData[bucket] = elementData[bucket].next;
            size--;
        } else {
            // check rest of list
            HashEntry current = elementData[bucket];
            while (current.next != null && current.next.data != value) {
                current = current.next;
            }

            // if the element is found, remove it
            if (current.next != null) {
                current.next = current.next.next;
                size--;
            }
        }
    }
}
```

Removing from a hash table that uses probing has different complications. If we start with the hash table from Figure 18.5 and decide to remove the value `1`, we can't just replace it with a blank value, because there is a chain of other elements (`91` and `71`) that probed to the following indexes 2 and 3. If a later `contains(71)` call is made on the set, the code might mistakenly think that bucket 1 is empty and therefore incorrectly decide that `71` is not contained in the set. What is often done instead is to replace the removed value with a special marker value to indicate that a removal occurred. That way when `contains` or `add` is called later, it can recognize that value and realize that other values might follow the removed bucket. Flagging buckets with this special marker is more efficient than trying to shift values back in the hash table. Figure 18.7 shows an example removal of the value `1`.

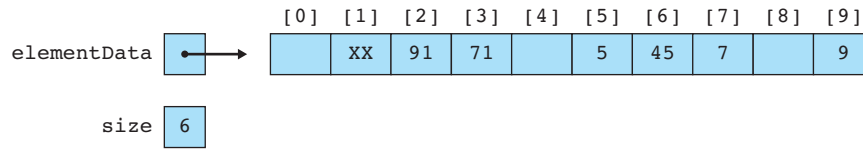

Figure 18.7 Removal of an element when probing is used