# Chapter 18
# 18.1: Hashing

- Array Set Implementations
- Hash Functions and Hash Tables
- Collisions
- Rehashing
- Hashing Non-Integer Data
- Hash Map Implementation

## Introduction

In this chapter we will explore the implementation of two powerful data structures called hash tables and heaps. Both of these structures are used in Java's collections framework to implement collections such as `HashSet`, `HashMap`, and `PriorityQueue`. These clever data structures allow for efficient operations such as adding, removing, searching, and ordering elements.

Upon completion of this chapter you will have seen all of the major implementation strategies used in the primary collections in the Java collection framework: arrays (`ArrayList`, `Stack`), linked lists (`LinkedList`), binary trees (`TreeSet`, `TreeMap`), hash tables (`HashSet`, `HashMap`), and heaps (`PriorityQueue`).

## 18.1 Hashing

*Hashing* is a technique for efficiently mapping data elements to indexes in an array so that they can be added, removed, and searched very quickly. Hashing is the underlying technique used to implement the `HashSet` and `HashMap` classes. Hashing makes it possible to have a collection that can add, remove, and search for data elements in a constant amount of time (also called O(1), as discussed in Chapter 13), making it extremely fast for many common use cases.

### Array Set Implementations

Suppose we want to store a set of integers using an array as the underlying data structure and that the most common operations we want to enable are insertion (`add`), deletion (`remove`), and testing for membership (`contains`). Further suppose that the order in which the elements are stored does not matter, so long as we can implement all of these operations efficiently. The key concern is that the structure should be fast for these three operations.

We could follow the same general idea as `ArrayIntList`, using an unfilled array and a `size` field to represent the data in the set:

```
public class ArrayIntSet {
    private int[] elementData;
    private int size;

    // Constructs an empty set.
    public ArrayIntSet() {
        elementData = new int[10];
        size = 0;
    }
    ...
```

As each element is inserted, it is stored at the next available index and the `size` field is incremented. So if the array's length is 10 and the client adds the values 7, 5, 1, and 9, the set would have the internal state shown in Figure 18.1. The blank array cells actually contain 0s, but they are unimportant because they are past the size.

Is this a good way to implement a set? Think about how the common operations would be written. Adding an element is efficient (O(1)); simply place it in the next free slot and increment the `size` field. Removal is an inefficient linear search through the array for the value to remove. Once the value is found, since order is unimportant, you can quickly remove it by replacing it by the last element in the array, rather than having to shift any following elements left by one index; but the search to find it is still slow. We might have to examine all of the elements (on average, about half of them), so removal is O(*N*). Testing for membership is also slow, a linear search over all elements to try to find the value of interest. To figure out whether the set contains the value 9, we must look at the 7, 5, 1, and 9 before it is found.

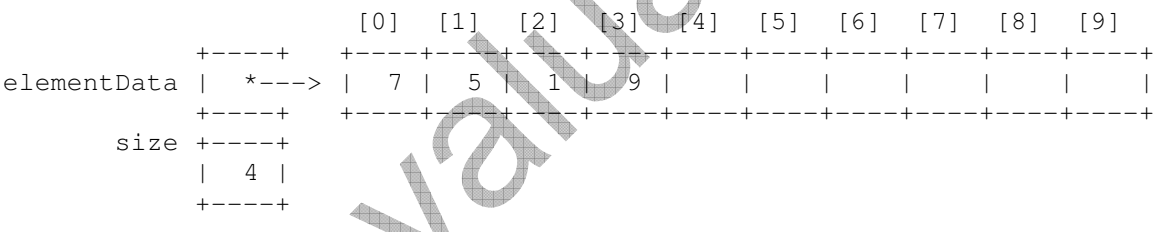


**Figure 18.1 Representing a set as an unfilled, unsorted array**

Removal and searching from the unsorted array are slow because there's no particular way to guess where the element value will be found, if it is in the array at all. But with some creativity we can improve upon this situation. We've seen some strategies that take advantage of sorted data, such as the binary search algorithm. Binary search does not need to examine every index of the array to find an element; at each step the algorithm eliminates half of the remaining search space, resulting in a O(log *N*) runtime.

We could base our set on a sorted array, as shown in Figure 18.2. As elements are added to this sorted array, they must be placed into the proper indexes to maintain sorted order. So the implementation of insertion or removal requires two steps: binary searching for the right index for the element, then shifting any following elements left or right. For example, if the value 2 is added, the 5, 7, and 9 must be shifted right by one index to make room for it.

The sorted array structure would be faster to search because we could implement our `contains` method as a binary search. Adding and removing would still be O(*N*) in the worst case because

of the shifting. Still, this is arguably a better implementation than the unsorted array if the primary concern is the speed of the `contains` operation.

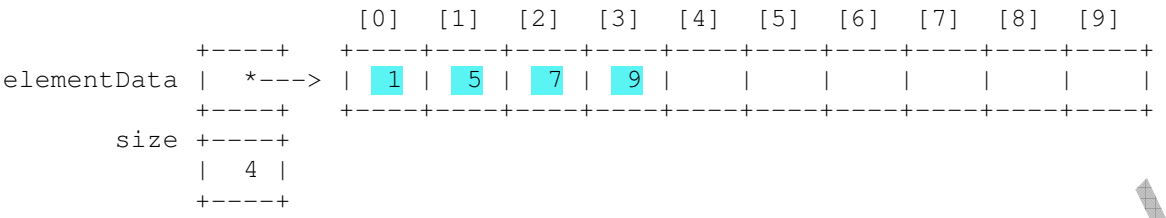


**Figure 18.2 Representing a set as an unfilled, sorted array**

## Hash Functions and Hash Tables

Implementing a set using a sorted array is a step up from an unsorted one, but it's possible to do better. Let's explore how to use a special mapping strategy called a hash function to organize the elements of our set and search it quickly.

Recall that the order of the elements doesn't matter, as long as the set can add, remove, and find elements. The elements can be placed anywhere, as long as they can be found later. Here's an odd but powerful idea: What if we stored element value *k* at index *k*? For example, if you tell the set to add the value 5, store it at index 5. If we used this strange technique, the set storing 7, 5, 1, and 9 would have the structure shown in Figure 18.3.

If we stored our set data using this technique, the three basic set operations become extremely efficient to implement. Inserting a value *k* simply involves going to index *k* and storing *k* there; no searching or shifting is needed. Removing the value *k* requires going to index *k* and changing the value back to 0. Testing for membership of a value *k* (`contains`) requires looking at index *k* to see whether *k* is stored there; if so, the value is part of the set, and if not, it isn't.

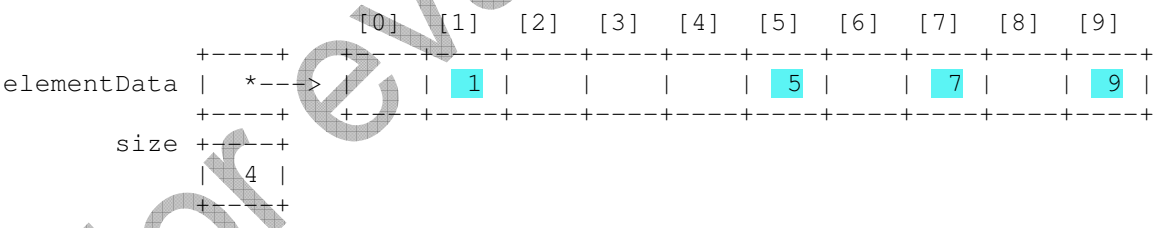


**Figure 18.3 Storing set elements at corresponding indexes**

You may already be thinking of some problems with this implementation strategy. What if the client tries to add a value that is outside the range 0 to 9, such as 23? We could start with a larger array; if `elementData` has length 100, our 23 will fit nicely. But no matter how large the array becomes, the client can always add a larger integer outside its bounds. Negative numbers also pose a problem because arrays never have negative indexes.

To get around these issues, let's patch our storage technique. Instead of always storing element value *k* at index *k*, we'll limit *k*'s value by modding it by the array capacity. So if the array length is 10, the value 23 would be inserted at index 3 because 23 % 10 equals 3. To fix negative

numbers, we'll take the absolute value of *k* and apply the same technique. So the value -58 would be inserted at index 8.

Figure 18.4 shows what the array would look like after the insertion of 23 and -58 using our new mod technique. The set still has the same speedy runtime for the common operations because each element still has a single index into which it will be placed.

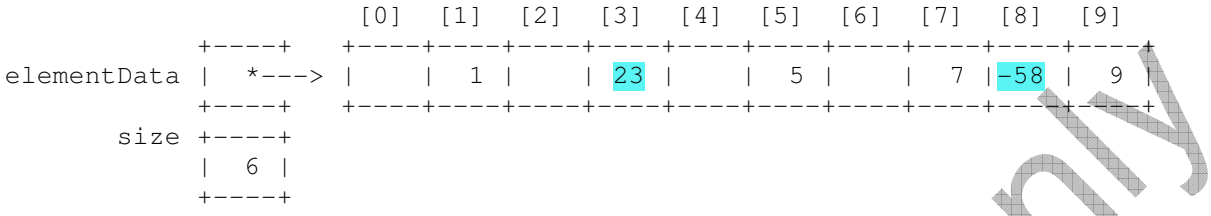


**Figure 18.4 Use of hash function with wrap-around**

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* that involve 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.

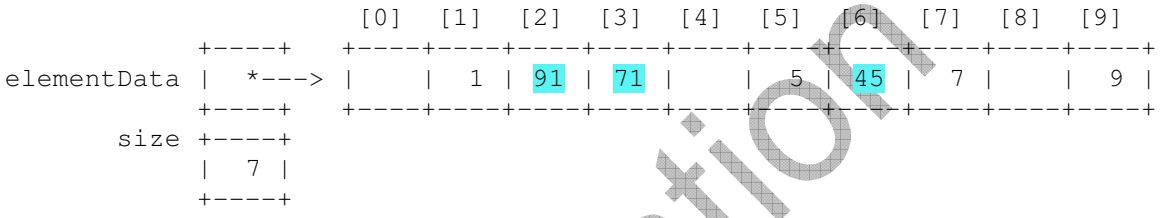


**Figure 18.5 Hash collisions resolved by linear probing**

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 collision issues 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, of course. If the client tries to add an eleventh element, the array must be resized. Resizing a hash table is a non-trivial 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.

> **separate chaining**
> Resolving hash collision issues 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.

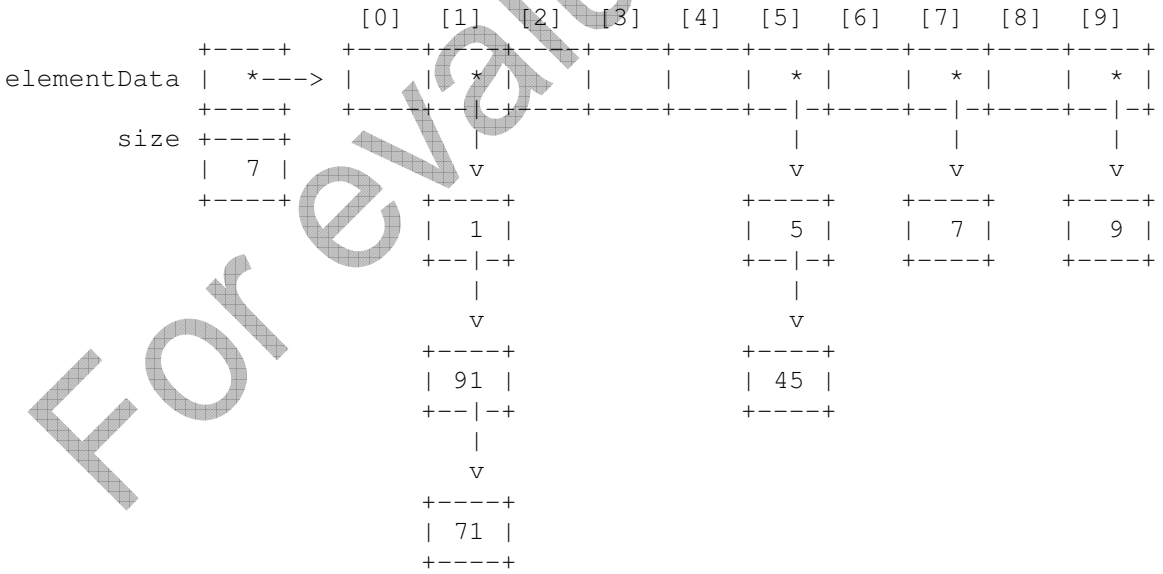


**Figure 18.6 Hash collisions resolved by separate chaining**

We'll implement the lists of elements as lists of node objects, similarly 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 {
    public int data;
    public 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.

```
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.data == value) {
                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. So 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` are called later, they 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.

We will focus on our implementation that uses separate chaining rather than showing the code for the probing version. You can find both versions on our web site at http://buildingjavaprograms.com.

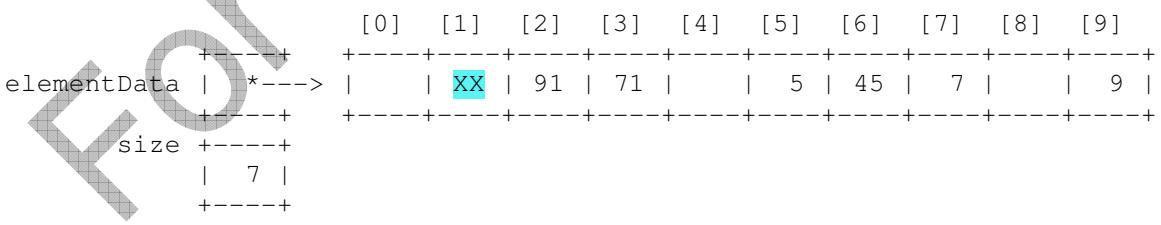


**Figure 18.7 Removal of an element when probing is used**

There are other ways of resolving collisions. For example, if the hash function results in a conflict, a secondary hash function can be used to find a new hash bucket. This is sometimes called *double hashing*. Double hashing and other methods of collision resolution are outside the scope of this chapter.

## Rehashing

When a hash table becomes too full, it can be enlarged to give it additional capacity for storing elements. Consider the full hash table in Figure 18.8 that uses linear probing to resolve collisions. In addition to the elements previously added, the client has inserted 80, 63, and 57, causing a completely full hash table. When the client tries to add an eleventh value, the hash table must enlarge to make room for it.

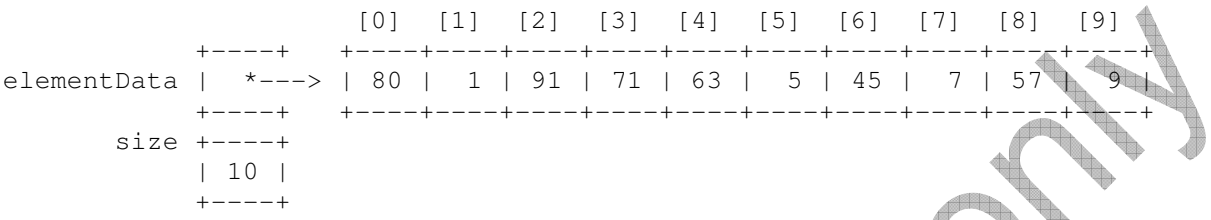


**Figure 18.8 Full hash table with linear probing**

Resizing a hash table is not as simple as creating a larger array and copying each element over into it. The reason is because the element values may not have the same hash function result in the new larger array. For example, the value 91 maps to index 1 in the array of length 10 because 91 % 10 equals 1. But the same value 91 maps to index 11 in an array of length 20 because 91 % 20 equals 11. Therefore when resizing a hash table array, it is important to reprocess each element using the hash function and place it into the new appropriate index. This process is called *rehashing*.

Suppose the client tries to add the value 35 to the full hash table. Figure 18.9 shows a properly resized and rehashed table. Notice that 91, 71, 35, and 57 now sit in the second half of the array because their hash function results fall in that range. The value 63 also gets to sit in its preferred index of 3 because the element 71 that previously collided with it has moved to index 12. The rehashing also helps to decrease clustering; now the longest chain of consecutively occupied buckets is three.

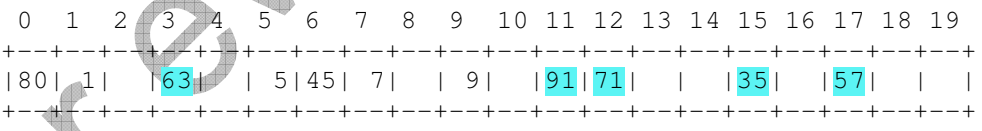


**Figure 18.9 Enlarged hash table after rehashing**

**rehashing**
Resizing a hash table to increase its capacity and enabling it to store more elements, or store them more efficiently.

Hash tables that use separate chaining can also be rehashed, but the issues are slightly different. Technically a chained hash table never becomes completely "full" in the sense of becoming unable to add any more elements, because a new element can always be added to the list at a given index. But if the lists at each index become too long, the set operations become too slow. So it is still important to enlarge the hash table as its size grows. Figure 18.10 shows a chained table with the same elements as in Figure 18.8.

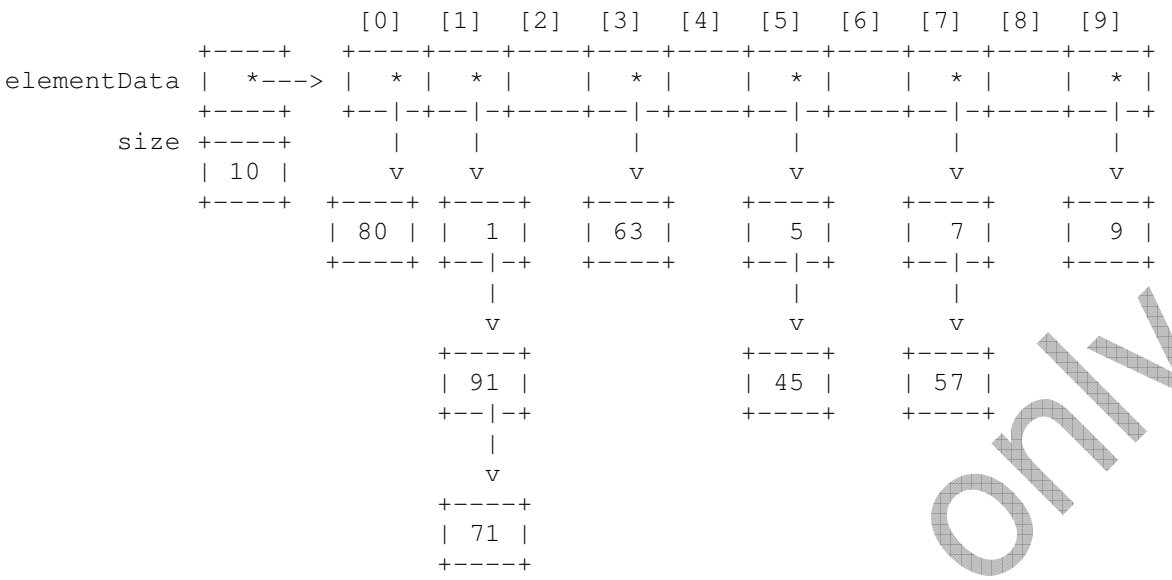


**Figure 18.10 "Full" hash table with separate chaining**

Figure 18.11 shows the same hash table after a rehashing to length 20 and the insertion of value 35. The values 91, 71, 35, and 57 are now stored in lists in the second half of the table.

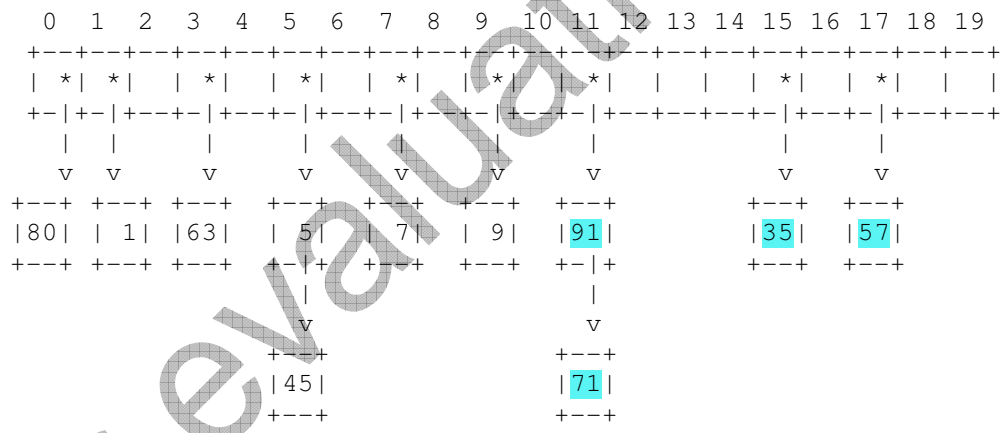


**Figure 18.11 Enlarged chained hash table after rehashing**

In our examples we chose to resize the hash table once its size exceeded its capacity. But most hash tables do not wait so long to rehash, because a nearly or entirely full table starts to exhibit poor performance. A better alternative is to monitor the table's size relative to the array capacity, also called the *load factor* of the table, and resize once this ratio exceeds some threshold. The load factor is fairly easy to compute:

```
private double loadFactor() {
    return (double) size / elementData.length;
}
```

The real Java `HashSet` uses a default load factor of 0.75; in other words, once the number of elements is three fourths as large as the capacity, the table is enlarged. We can incorporate this logic into our own hash set with a constant and a check at the start of the `add` method:

```
public class HashIntSet {
    private static final double MAX_LOAD_FACTOR = 0.75;
    private HashEntry[] elementData;
    private int size;
    ...

    public void add(int value) {
        if (!contains(value)) {
            if (loadFactor() >= MAX_LOAD_FACTOR) {
                rehash();
            }
        ...
```

The simplest way to implement the actual rehashing operation is to replace the set's internal hash table with a new larger one, then loop over the old table and re-add all of its contents back into the new hash table. We can call the set's own `add` method to re-insert all of the data to avoid duplicating code:

```
private void rehash() {
    // replace element data array with a larger empty version
    HashEntry[] oldElementData = elementData;
    elementData = new HashEntry[2 * oldElementData.length];
    size = 0;

    // re-add all of the old data into the new array
    for (int i = 0; i < oldElementData.length; i++) {
        HashEntry current = oldElementData[i];
        while (current != null) {
            add(current.data);
            current = current.next;
        }
    }
}
```

## Hashing Non-Integer Data

Our hash set works well for storing integer data, but we have not yet discussed how to apply hashing techniques to non-integer data. The key idea is still the same: We need a way of mapping from values to preferred indexes at which to store them. It is possible to come up with hash functions for other kinds of data that convert them into integers. For example, we could convert a string into an integer by adding the integer ASCII values of its characters in some way.

The idea of hashing, and the importance of all kinds of data being able to be stored in a hash table, was close to the hearts of the original designers of Java. Recall that every Java class has an ancestor named `Object` that contains various basic behavior that is common to every object. One of the methods of the `Object` class is named `hashCode`. You can call `hashCode` on any object to convert the object into an integer for use in a hash table. A modified version of our hash function that works for any type of objects is the following:

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

According to the official Java documentation for the `hashCode` method, a proper hash function must be consistent. That is, when called multiple times on an object whose state is not changing, it should return the same value each time. Similarly, when `hashCode` is called on two objects with equivalent state (such that `equals` would return `true` for the two objects), it should return the same value.

In general it is also desirable for `hashCode` to return unequal values when two objects have different state. But it is not mandatory for unequal objects to always have different hash codes. In fact, it is impossible to guarantee such a thing; for example, there are only four billion unique integer values, but there are infinitely many unique strings, so some strings must naturally have the same hash code as others. Unequal objects having equal hash codes does not break the hash table, but it does mean that those objects will hash to the same index and will collide, so they will be placed into the same list if separate chaining is used on the hash table.

The default version of `hashCode` from the `Object` class computes an integer based on the memory address of the object. This function satisfies the consistency requirement, but it is not ideal because it does not take into account the state of each object. For this reason many classes override `hashCode` with their own versions that compute an integer by somehow combining the object's state. Good hash functions try to do their best about spreading the hash values over a wide range of numbers to minimize conflicts.

When writing your own class, if your class has an `equals` method, you are expected to also provide a `hashCode` method to override the one provided by the `Object` superclass. The reason is because objects with "equal" state are supposed to have equal hash codes, and this would not be guaranteed by the version of `hashCode` inherited from class `Object`. Your `hashCode` method should combine the object's state in some way into a single integer and return it. The state can be combined in any way as long as it consistently produces the same integer for the same state.

The following is a possible `hashCode` method for the `Point` class implemented in Chapter 8. The code adds the `x` and `y` values but scales the `y` value up by a large number to spread out the space of integers returned. This is to avoid collisions between points like (4, 7) and (7, 4). Simply adding `x + y` and returning the result would also be correct but would not distribute the integer codes as much.

```
// a possible hashCode method for Ch. 8's Point class
public class Point {
    private int x;
    private int y;
    ...
    public int hashCode() {
        return 31337 * y + x;
    }
```

Many classes in the Java class libraries have their own `hashCode` methods. For example, the `String` class code has a `hashCode` method whose code is roughly the following. The code seems odd in that it multiplies by 31 at each pass through the loop. If the characters were simply added,

there would be conflicts between various kinds of strings. For example, strings of similar length are all clustered within a small range of hash codes, and strings that are anagrams, such as "file" and "life", always collide. Multiplying the result by some number at each pass spreads out the space of codes and reduces these potential collisions.

```
// the hashCode method inside Java's String class
public int hashCode() {
    int hash = 0;
    for (int i = 0; i < this.length(); i++) {
        hash = 31 * hash + this.charAt(i);
    }
    return hash;
}
```

If you are writing a `hashCode` method for a class that has fields that are not integers, it is less clear how to combine the fields' state into a single hash code integer. Remember that if the fields are objects, they have their own `hashCode` methods that you can call. And primitives like `double` can be converted into their object wrapper equivalents, such as `Double`, and you can call `hashCode` on that. Here is an example class `BankAccount` with non-integer fields and a possible `hashCode` method based on its state. Notice that we again multiply the various fields by large integer constants to spread out the results of the function.

```
// a possible hashCode method for a BankAccount class
public class BankAccount {
    private int id;
    private String name;
    private double balance;
    ...
    public int hashCode() {
        return id +
               31 * name.hashCode() +
               16581 * new Double(balance).hashCode();
    }
```

If we want our set to be able to store any kind of object as its data, we can convert our `HashIntSet` into a generic `HashSet`, giving it a type parameter `E` for its elements. This requires small changes throughout the code, such as changing methods to accept parameters of type `E` rather than `int`, and comparing values using `equals` rather than `==`. The process is very similar to our conversions of `ArrayIntList` and `LinkedIntList` to generic collections in previous chapters. The complete generic version of the class is not shown, but a few of the relevant lines to change are found in the following abbreviated code:

```
public class HashSet<E> {
    ...
    public void add(E value) { ... }
    public boolean contains(E value) {
        ...
        while (current != null) {
            if (current.data.equals(value)) { ...
    }
    public void remove(E value) { ... }
    ...
    private class HashEntry {
        public E data;
        public HashEntry next;
        ...
    }
}
```

## Hash Map Implementation

The ideas shown so far form the core of the implementation of the `HashSet` class. The `HashMap` has a very similar internal structure. Recall that a map stores key/value pairs rather than simple element values. A `HashMap` is still implemented using a hash table, but the hash entry objects in each linked lists are changed to store a pair of data fields, representing one key/value pair.

```
public class HashEntry {
    private Object key;
    private Object value;
    private HashEntry next;
    ...
}
```

Suppose a map is created with integers as the type for its keys and values, and the following pairs are added to it:

```
mymap.put(1, 37);
mymap.put(39, -2);
mymap.put(25, 44);
mymap.put(-5, 0);
```

Figure 18.12 shows the internal state of the hash table containing these pairs. Each index stores a list of `HashEntry` objects.

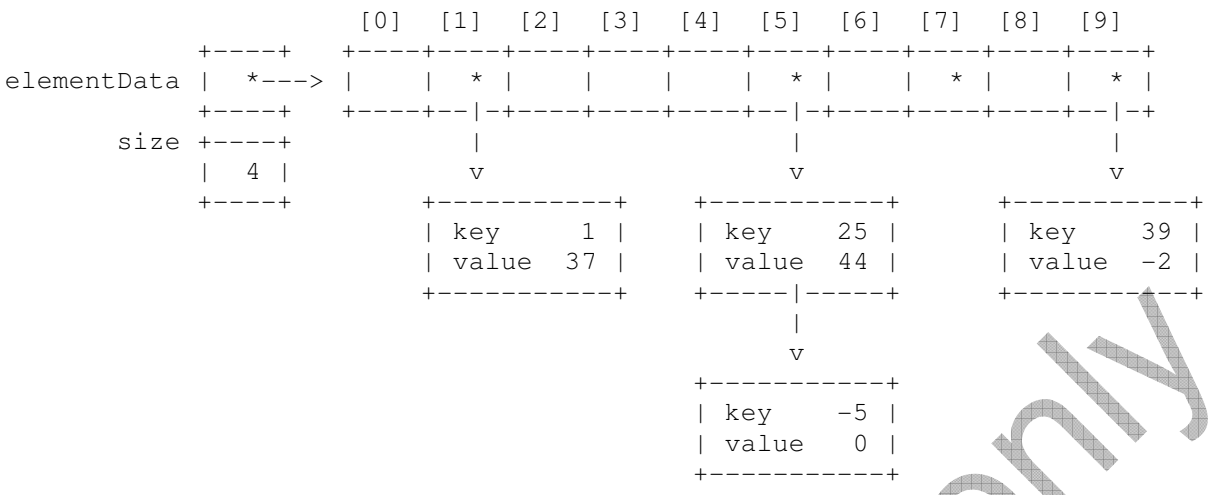


**Figure 18.12 Hash map**

To be able to accept any type of data for the keys and values, most hash map implementations are generic classes that accept a pair of type parameters, `K` for the key type and `V` for the value type. The hash map therefore has the following class header:

```
public class HashMap<K, V> {
    ...
    private class HashEntry {
        private K key;
        private V value;
        private HashEntry next;
        ...
    }
}
```