14.1 A Hashing

In all of our discussions of the implementations of collections, we have proceeded with one of two assumptions about the order of elements in a collection:

- Order is determined by the order in which elements are added to and/or removed from our collection, as in the case of stacks, queues, unordered lists, and indexed lists.
- Order is determined by comparing the values of the elements (or some key component of the elements) to be stored in the collection, as in the case of ordered lists and binary search trees.

In this chapter, we will explore the concept of *hashing*, which means that the order—and, more specifically, the location of an item within the collection—is determined by some function of the value of the element to be stored, or some function of a key value of the element to be stored. In hashing, elements are stored in a *hash table*, with their location in the table determined by a *hashing function*. Each location in the table may be referred to as a *cell* or a *bucket*. We will discuss hashing functions further in Section 14.2. We will discuss implementation strategies and algorithms, and we will leave the implementations as programming projects.

KEY CONCEPT
In hashing, elements are stored in a hash table, with their location in the table determined by a hashing function.

Consider a simple example where we create an array that will hold 26 elements. Wishing to store names in our array, we create a hashing function that equates each name to the position in the array associated with the first letter of the name (e.g., a first letter of A would be mapped to position 0 of the array, a first letter of D would be mapped to position 3 of the array, and so on). Figure 14.1 illustrates this scenario after several names have been added.

Notice that unlike our earlier implementations of collections, using a hashing approach results in the access time to a particular element being independent of the number of elements in the table. This means that all of the operations on an element of a hash table should be O(1). This is the result of no longer having to do comparisons to find a particular element or to locate the appropriate position for a given element. Using hashing, we simply calculate where a particular element should be.

KEY CONCEPT
The situation where two elements or keys map to the same location in the table is called a collision.

However, this efficiency is only fully realized if each element maps to a unique position in the table. Consider our example from Figure 14.1. What will happen if we attempt to store the name "Ann" and the name "Andrew"? This situation, where two elements or keys map to the same location in the table, is called a *collision*. We will discuss how to resolve collisions in Section 14.3.

KEY CONCEPT
A hashing function that maps each element to a unique position in the table is said to be a perfect hashing function.

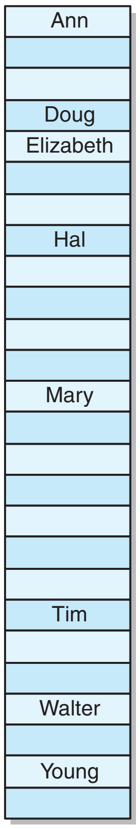

FIGURE 14.1 A simple hashing example

A hashing function that maps each element to a unique position in the table is said to be a *perfect hashing function*. Although it is possible in some situations to develop a perfect hashing function, a hashing function that does a good job of distributing the elements among the table positions will still result in constant time (O(1)) access to elements in the table and an improvement over our earlier algorithms that were either O(n) in the case of our linear approaches or O(log n) in the case of search trees.

Another issue surrounding hashing is the question of how large the table should be. If the data set is of known size and a perfect hashing function can be used, then we simply make the table the same size as the data set. If a perfect hashing function is not available or practical but the size of the data set is known, a good rule of thumb is to make the table 150 percent the size of the data set.

The third case is very common and far more interesting. What if we do not know the size of the data set? In this case, we depend on *dynamic resizing*. Dynamic resizing of a hash table involves creating a new hash table that is larger than, perhaps even twice as large as, the original, inserting all of the elements of the original table into the new table, and then discarding the original table. Deciding when to resize is also an interesting question. One possibility is to use the same method we used with our earlier array implementations and simply expand the table when it is full. However, it is the nature of hash tables that their performance seriously degrades as they become full. A better approach is to use a *load factor*. The load factor of a hash table is the percentage occupancy of the table at which the table will be resized. For example, if the load factor were set to 0.50, then the table would be resized each time it reached 50 percent capacity.

14.2 Hashing Functions

KEY CONCEPT
Extraction involves using only a part of the element's value or key to compute the location at which to store the element.

Although perfect hashing functions are possible if the data set is known, we do not need the hashing function to be perfect to get good performance from the hash table. Our goal is simply to develop a function that does a reasonably good job of distributing our elements in the table such that we avoid collisions. A reasonably good hashing function will still result in constant time access (O(1)) to our data set.

There are a variety of approaches to developing a hashing function for a particular data set. The method that we used in our example in the previous section is called *extraction*. Extraction involves using only a part of the element's value or key to compute the location at which to store the element. In our previous example, we simply extracted the first letter of a string and computed its value relative to the letter A.

Other examples of extraction would be to store phone numbers according to the last four digits or to store information about cars according to the first three characters of the license plate.

The Division Method

Creating a hashing function by *division* simply means we will use the remainder of the key divided by some positive integer p as the index for the given element. This function could be defined as follows:

```
Hashcode(key) = Math.abs(key)%p
```

This function will yield a result in the range from 0 to p–1. If we use our table size as p, we then have an index that maps directly to a location in the table.

Using a prime number p as the table size and the divisor helps provide a better distribution of keys to locations in the table.

For example, if our key value is 79 and our table size is 43, the division method would result in an index value of 36. The division method is very effective when dealing with an unknown set of key values.

The Folding Method

In the *folding method*, the key is divided into parts that are then combined or folded together to create an index into the table. This is done by first dividing the key into parts where each of the parts of the key will be the same length as the desired index, except possibly the last one. In the *shift folding method*, these parts are then added together to create the index. For example, if our key is the Social Security number 987-65-4321, we might divide this into three parts, 987, 654, and 321. Adding these together yields 1962. Assuming we are looking for a three-digit key, at this point we could use either division or extraction to get our index.

KEY CONCEPT
In the shift folding method, the parts of the key are added together to create the index.

A second possibility is *boundary folding*. There are a number of variations on this approach. However, generally, they involve reversing some of the parts of the key before adding. One variation on this approach is to imagine that the parts of the key are written side by side on a piece of paper and that the piece of paper is folded along the boundaries of the parts of the key. In this way, if we begin with the same key, 987-65-4321, we first divide it into parts, 987, 654, and 321. We then reverse every other part of the key, yielding 987, 456, and 321. Adding these together yields 1764 and once again we can proceed with either extraction or division to get our index. Other variations on folding use different algorithms to determine which parts of the key to reverse.

Folding may also be a useful method for building a hashing function for a key that is a string. One approach to this is to divide the string into substrings the same length (in bytes) as the desired index and then combine these strings using an *exclusive-or* function. This is also a useful way to convert a string to a number so that other methods, such as division, may be applied to strings.

The Mid-Square Method

In the *mid-square method*, the key is multiplied by itself, and then the extraction method is used to extract the appropriate number of digits from the middle of the squared result to serve as an index. The same "middle" digits must be chosen each time, to provide consistency. For example, if our key is 4321, we would multiply the key by itself, yielding 18671041. Assuming that we need a three-digit key, we might extract 671 or 710, depending upon how we construct our algorithm. It is also possible to extract bits instead of digits and then construct the index from the extracted bits.

The mid-square method may also be effectively used with strings by manipulating the binary representations of the characters in the string.

The Radix Transformation Method

In the *radix transformation method*, the key is transformed into another numeric base. For example, if our key is 23 in base 10, we might convert it to 32 in base 7. We then use the division method and divide the converted key by the table size and use the remainder as our index. Continuing our previous example, if our table size is 17, we would compute the function:

```
Hashcode(23) = Math.abs(32)%17
             = 15
```

The Digit Analysis Method

In the *digit analysis method*, the index is formed by extracting, and then manipulating, specific digits from the key. For example, if our key is 1234567, we might select the digits in positions 2 through 4, yielding 234, and then manipulate them to form our index. This manipulation can take many forms, including simply reversing the digits (yielding 432), performing a circular shift to the right (yielding 423), performing a circular shift to the left (yielding 342), swapping each pair of digits (yielding 324), or any number of other possibilities, including the methods we have already discussed. The goal is simply to provide a function that does a reasonable job of distributing keys to locations in the table.

The Length-Dependent Method

In the *length-dependent method*, the key and the length of the key are combined in some way to form either the index itself or an intermediate value that is then used with one of our other methods to form the index. For example, if our key is 8765, we might multiply the first two digits by the length and then divide by the last digit, yielding 69. If our table size is 43, we would then use the division method, resulting in an index of 26.

KEY CONCEPT
The length-dependent method and the mid-square method may also be effectively used with strings by manipulating the binary representations of the characters in the string.

The length-dependent method may also be effectively used with strings by manipulating the binary representations of the characters in the string.

Hashing Functions in the Java Language

The `java.lang.Object` class defines a method called `hashcode` that returns an integer based on the memory location of the object. This is generally not very useful. Classes that are derived from `Object` often override the inherited definition of `hashcode` to provide their own version. For example, the `String` and `Integer` classes define their own `hashcode` methods. These more specific `hashcode` functions can be very effective for hashing. By having the `hashcode` method defined in the `Object` class, all Java objects can be hashed. However, it is also possible, and often preferable, to define your own `hashcode` method for any class that you intend to store in a hash table.

KEY CONCEPT

Although Java provides a `hashcode` method for all objects, it is often preferable to define a specific hashing function for any particular class.

14.3 Resolving Collisions

If we are able to develop a perfect hashing function for a particular data set, then we do not need to concern ourselves with collisions, the situation where more than one element or key map to the same location in the table. However, when a perfect hashing function is not possible or practical, there are a number of ways to handle collisions. Similarly, if we are able to develop a perfect hashing function for a particular data set, then we do not need to concern ourselves with the size of the table. In this case, we will simply make the table the exact size of the data set. Otherwise, if the size of the data set is known, it is generally a good idea to set the initial size of the table to about 150 percent of the expected element count. If the size of the data set is not known, then dynamic resizing of the table becomes an issue.

Chaining

The *chaining method* for handling collisions simply treats the hash table conceptually as a table of collections rather than as a table of individual cells. Thus each cell is a pointer to the collection associated with that location in the table. Usually this internal collection is either an unordered list or an ordered list. Figure 14.2 illustrates this conceptual approach.

KEY CONCEPT

The chaining method for handling collisions simply treats the hash table conceptually as a table of collections rather than as a table of individual cells.

Chaining can be implemented in a variety of ways. One approach would be to make the array holding the table larger than the number of cells in the table and use the extra space as an overflow area to store the linked lists associated with each table location. In this method, each position in the array could store both an element (or a key) and the array index of the next

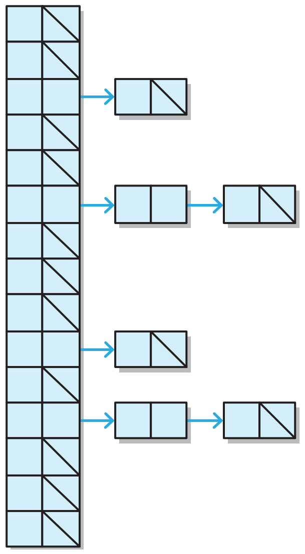

FIGURE 14.2 The chaining method of collision handling

element in its list. The first element mapped to a particular location in the table would actually be stored in that location. The next element mapped to that location would be stored in a free location in this overflow area, and the array index of this second element would be stored with the first element in the table. If a third element is mapped to the same location, the third element would also be stored in this overflow area and the index of third element would be stored with the second element. Figure 14.3 illustrates this strategy.

Note that, using this method, the table itself can never be full. However, if the table is implemented as an array, the array can become full, requiring a decision on whether to throw an exception or simply expand capacity. In our earlier collections, we chose to expand the capacity of the array. In this case, expanding the capacity of the array but leaving the embedded table the original size would have disastrous effects on efficiency. A more complete solution is to expand the array and expand the embedded table within the array. This will, however, require that all of the elements in the table be rehashed using the new table size. We will discuss the dynamic resizing of hash tables further in Section 14.5.

Using this method, the worst case is that our hashing function will not do a good job of distributing elements to locations in the table so that we end up with one linked list of n elements, or a small number of linked lists with roughly n/k elements each, where k is some relatively small constant. In this case, hash tables become O(n) for both insertions and searches. Thus you can see how important it is to develop a good hashing function.

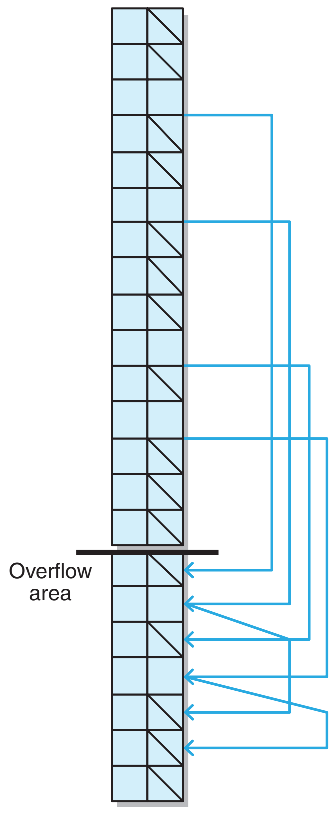

FIGURE 14.3 Chaining using an overflow area

A second method for implementing chaining is to use links. In this method, each cell or bucket in the hash table would be something like the `LinearNode` class used in earlier chapters to construct linked lists. In this way, as a second element is mapped to a particular bucket, we simply create a new `LinearNode`, set the `next` reference of the existing node to point to the new node, set the `element` reference of the new node to the element being inserted, and set the `next` reference of the new node to null. The result is an implementation model that looks exactly like the conceptual model shown in Figure 14.2.

A third method for implementing chaining is to literally make each position in the table a pointer to a collection. In this way, we could represent each position in the table with a list or perhaps even a more efficient collection (e.g., a balanced binary search tree), and this would improve our worst case. Keep in mind, however, that if our hashing function is doing a good job of distributing elements to locations in the table, this approach may incur a great deal of overhead while accomplishing very little improvement.

Open Addressing

The *open addressing method* for handling collisions looks for another open position in the table other than the one to which the element is originally hashed. There are a variety of methods to find another available location in the table. We will examine three of these methods: linear probing, quadratic probing, and double hashing.

KEY CONCEPT

The open addressing method for handling collisions looks for another open position in the table other than the one to which the element is originally hashed.

The simplest of these methods is *linear probing*. In linear probing, if an element hashes to position p and position p is already occupied, we simply try position (p+1)%s, where s is the size of the table. If position (p+1)%s is already occupied, we try position (p+2)%s, and so on until either we find an open position or we find ourselves back at the original position. If we find an open position, we insert the new element. What to do if we do not find an open position is a design decision when creating a hash table. As we have discussed previously, one possibility is to throw an exception if the table is full. A second possibility is to expand the capacity of the table and rehash the existing entries.

The problem with linear probing is that it tends to create clusters of filled positions within the table, and these clusters then affect the performance of insertions and searches. Figure 14.4 illustrates the linear probing method and the creation of a cluster using our earlier hashing function of extracting the first character of the string.

In this example, Ann was entered, followed by Andrew. Because Ann already occupied position 0 of the array, Andrew was placed in position 1. Later, Bob was entered. Because Andrew already occupied position 1, Bob was placed in the next open position, which was position 2. Doug and Elizabeth were already in the table by the time Betty arrived, thus Betty could not be placed in position 1, 2, 3, or 4 and was placed in the next open position, position 5. After Barbara, Hal, and Bill were added, we find that there is now a nine-location cluster at the front of the table, which will continue to grow as more names are added. Thus we see that linear probing may not be the best approach.

A second form of the open addressing method is *quadratic probing*. Using quadratic probing, instead of using a linear approach, once we have a collision, we follow a formula such as

$$\texttt{newhashcode(x)} = \texttt{hashcode(x)} + (-1)^{i-1}((i + 1)/2)^2$$

for i in the range 1 to s–1 where s is the table size.

The result of this formula is the search sequence p, p + 1, p − 1, p + 4, p − 4, p + 9, p − 9, ... Of course, this new hash code is then put through the division method to keep it within the table range. As with linear probing, the same possibility exists that we will eventually get back to the original hash code without having found an open position in which to insert. This “full” condition can be

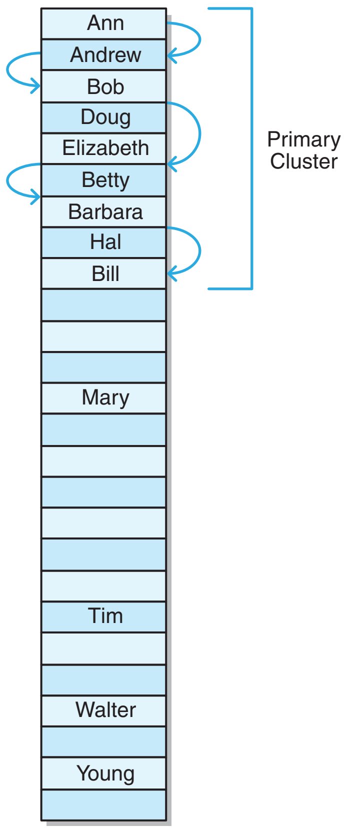

FIGURE 14.4 Open addressing using linear probing

handled in all of the same ways that we described for chaining and linear probing. The benefit of the quadratic probing method is that it does not have as strong a tendency toward clustering as does linear probing. Figure 14.5 illustrates quadratic probing for the same key set and hashing function that we used in Figure 14.4. Notice that after the same data has been entered, we still have a cluster at the front of the table. However, this cluster occupies only six buckets instead of the nine-bucket cluster created by linear probing.

A third form of the open addressing method is *double hashing*. Using the double hashing method, we will resolve collisions by providing a secondary hashing function to be used when the primary hashing function results in a collision. For example, if a key x hashes to a position p that is already occupied, then the next position p′ that we will try will be

```
p' = p + secondaryhashcode(x)
```

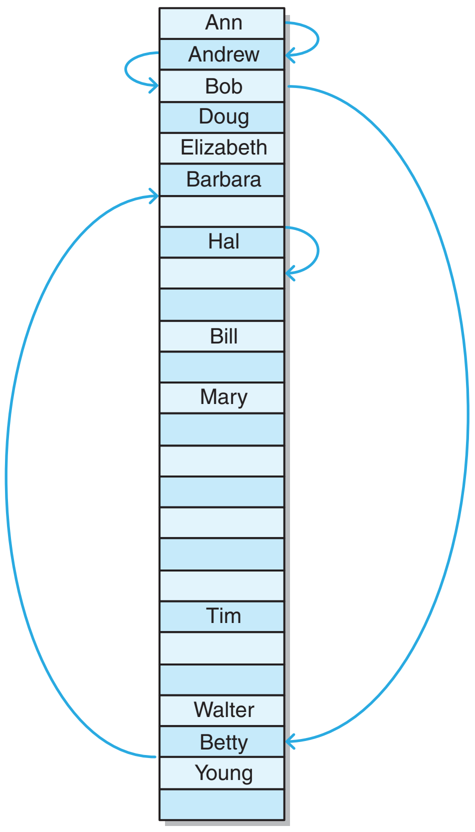

FIGURE 14.5 Open addressing using quadratic probing

If this new position is also occupied, then we look to position

```
p″ = p + 2 * secondaryhashcode(x)
```

We continue searching this way, of course using the division method to maintain our index within the bounds of the table, until an open position is found. This method, while somewhat more costly because of the introduction of an additional function, tends to further reduce clustering beyond the improvement gained by quadratic probing. Figure 14.6 illustrates this approach, again using the same key set and hashing function from our previous examples. For this example, the secondary hashing function is the length of the string. Notice that with the same data, we no longer have a cluster at the front of the table. However, we have

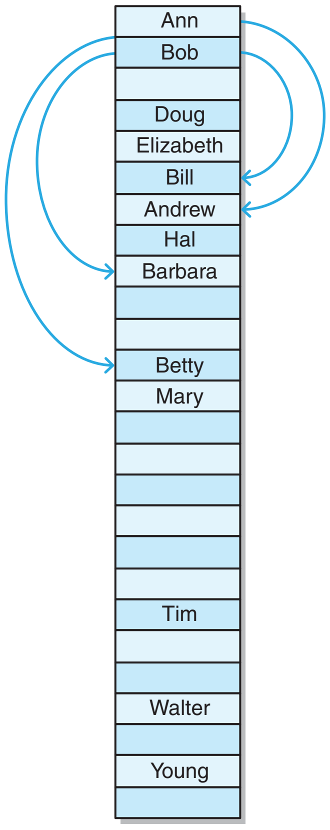

FIGURE 14.6 Open addressing using double hashing

developed a six-bucket cluster from Doug through Barbara. The advantage of double hashing, however, is that even after a cluster has been created, it will tend to grow more slowly than it would if we were using linear probing or even quadratic probing.

14.4 Deleting Elements from a Hash Table

Thus far, our discussion has centered on the efficiency of insertion of and searching for elements in a hash table. What happens if we remove an element from a hash table? The answer to this question depends upon which implementation we have chosen.

Deleting from a Chained Implementation

If we have chosen to implement our hash table using a chained implementation and an array with an overflow area, then removing an element falls into one of five cases:

Case 1 The element we are attempting to remove is the only one mapped to the particular location in the table. In this case, we simply remove the element by setting the table position to null.

Case 2 The element we are attempting to remove is stored in the table (not in the overflow area) but has an index into the overflow area for the next element at the same position. In this case, we replace the element and the next index value in the table with the element and next index value of the array position pointed to by the element to be removed. We then also must set the position in the overflow area to null and add it back to whatever mechanism we are using to maintain a list of free positions.

Case 3 The element we are attempting to remove is at the end of the list of elements stored at that location in the table. In this case, we set its position in the overflow area to null, and we set the next index value of the previous element in the list to null as well. We then also must set the position in the overflow area to null and add it back to whatever mechanism we are using to maintain a list of free positions.

Case 4 The element we are attempting to remove is in the middle of the list of elements stored at that location in the table. In this case, we set its position in the overflow area to null, and we set the next index value of the previous element in the list to the next index value of the element being removed. We then also must add it back to whatever mechanism we are using to maintain a list of free positions.

Case 5 The element we are attempting to remove is not in the list. In this case, we throw an `ElementNotFoundException`.

If we have chosen to implement our hash table using a chained implementation where each element in the table is a collection, then we simply remove the target element from the collection.

Deleting from an Open Addressing Implementation

If we have chosen to implement our hash table using an open addressing implementation, then deletion creates more of a challenge. Consider the example in Figure 14.7. Notice that elements "Ann," "Andrew," and "Amy" all mapped to the same location in the table and the collision was resolved using linear probing. What happens if we now remove "Andrew"? If we then search for "Amy" we will not find that element because the search will find "Ann" and then follow the linear probing rule to look in the next position, find it null, and return an exception.

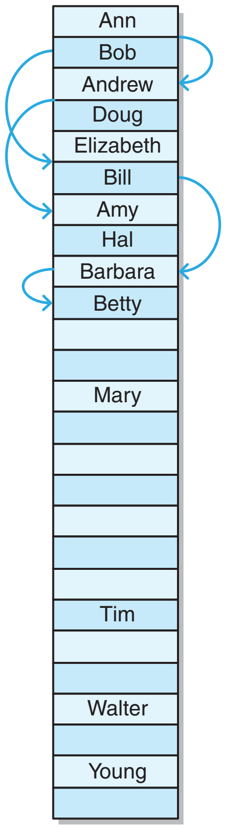

FIGURE 14.7 Open addressing and deletion

The solution to this problem is to mark items as deleted but not actually remove them from the table until some future point when the deleted element is overwritten by a new inserted element or the entire table is rehashed, either because it is being expanded or because we have reached some predetermined threshold for the percentage of deleted records in the table. This means that we will need to add a `boolean` flag to each node in the table and modify all of our algorithms to test and/or manipulate that flag.

14.5 Hash Tables in the Java Collections API

The Java Collections API provides seven implementations of hashing: `Hashtable`, `HashMap`, `HashSet`, `IdentityHashMap`, `LinkedHashSet`, `LinkedHashMap`, and `WeakHashMap`. To understand these different solutions we must first remind ourselves of the distinction between a *set* and a *map* in the Java Collections API as well as some of our other pertinent definitions.

KEY CONCEPT

The load factor is the maximum percentage occupancy allowed in the hash table before it is resized.

A *set* is a collection of objects where in order to find an object, we must have an exact copy of the object for which we are looking. A *map*, on the other hand, is a collection that stores key-value pairs so that, given the key, we can find the associated value.

Another definition that will be useful to us as we explore the Java Collections API implementations of hashing is that of a *load factor*. The load factor, as stated earlier, is the maximum percentage occupancy allowed in the hash table before it is resized. For the implementations that we are going to discuss here, the default is 0.75. Thus, using this default, when one of these implementations becomes 75 percent full, a new hash table is created that is twice the size of the current one, and then all of the elements from the current table are inserted into the new table. The load factor of these implementations can be altered when the table is created.

All of these implementations rely on the `hashcode` method of the object being stored to return an integer. This integer is then processed using the division method (using the table size) to produce an index within the bounds of the table. As stated earlier, the best practice is to define your own `hashcode` method for any class that you intend to store in a hash table.

Let's look at each of these implementations.

The `Hashtable` Class

The `Hashtable` implementation of hashing is the oldest of the implementations in the Java Collections API. In fact, it predates the Collections API and was modified in version 1.2 to implement the `Map` interface so that it would become a part of the Collections API. Unlike the newer Java Collections implementations, `Hashtable` is synchronized. Figure 14.8 shows the operations for the `Hashtable` class.

Creation of a `Hashtable` requires two parameters: initial capacity (with a default of 11) and load factor (with a default of 0.75). Capacity refers to the number of cells or locations in the initial table. Load factor is, as we described earlier, the maximum percentage occupancy allowed in the hash table before it is resized. `Hashtable` uses the chaining method for resolving collisions.

The `Hashtable` class is a legacy class that will be most useful if you are connecting to legacy code or require synchronization. Otherwise, it is preferable to use the `HashMap` class.

Return Value	Method	Description
	`Hashtable()`	Constructs a new, empty hash table with a default initial capacity (11) and load factor, which is 0.75.
	`Hashtable(int initialCapacity)`	Constructs a new, empty hash table with the specified initial capacity and default load factor, which is 0.75.
	`Hashtable(int initialCapacity, float loadFactor)`	Constructs a new, empty hash table with the specified initial capacity and the specified load factor.
	`Hashtable (Map t)`	Constructs a new hash table with the same mappings as the given `Map`.
`void`	`clear()`	Clears this hash table so that it contains no keys.
`Object`	`clone()`	Creates a shallow copy of this hash table.
`boolean`	`contains(Object value)`	Tests if some key maps into the specified value in this hash table.
`boolean`	`containsKey(Object key)`	Tests if the specified object is a key in this hash table.
`boolean`	`containsValue (Object value)`	Returns true if this hash table maps one or more keys to this value.
`Enumeration`	`elements()`	Returns an enumeration of the values in this hash table.
`Set`	`entrySet()`	Returns a `Set` view of the entries contained in this hash table.
`boolean`	`equals(Object o)`	Compares the specified `Object` with this `Map` for equality, as per the definition in the `Map` interface.
`Object`	`get(Object key)`	Returns the value to which the specified key is mapped in this hash table.
`int`	`hashCode()`	Returns the hash code value for this `Map` as per the definition in the `Map` interface.
`boolean`	`isEmpty()`	Tests if this hash table maps no keys to values.
`Enumeration`	`keys()`	Returns an enumeration of the keys in this hash table.
`Set`	`keysSet()`	Returns a `Set` view of the keys contained in this hash table.
`Object`	`put(Object key Object value)`	Maps the specified key to the specified value in this hash table.
`void`	`putAll(Map t)`	Copies all of the mappings from the specified `Map` to this hash table. These mappings will replace any mappings that this hash table had for any of the keys currently in the specified `Map`.
`protected void`	`rehash()`	Increases the capacity of and internally reorganizes this hash table, in order to accommodate and access its entries more efficiently.

FIGURE 14.8 Operations on the `Hashtable` class

`Object`	`remove(Object key)`	Removes the key (and its corresponding value) from this hash table.
`int`	`size()`	Returns the number of keys in this hash table.
`String`	`toString()`	Returns a string representation of this hash table object in the form of a set of entries, enclosed in braces and separated by the ASCII characters comma and space.
`Collection`	`values()`	Returns a `Collection` view of the values contained in this hash table.

FIGURE 14.8 *Continued*

The `HashSet` Class

The `HashSet` class implements the `Set` interface using a hash table. The `HashSet` class, like most of the Java Collections API implementations of hashing, uses chaining to resolve collisions (each table position effectively being a linked list). The `HashSet` implementation does not guarantee the order of the set on iteration and does not guarantee that the order will remain constant over time. This is because the iterator simply steps through the table in order. Because the hashing function will somewhat randomly distribute the elements to table positions, order cannot be guaranteed. Further, if the table is expanded, all of the elements are rehashed relative to the new table size, and the order may change.

Like the `Hashtable` class, the `HashSet` class also requires two parameters: initial capacity and load factor. The default for the load factor is the same as it is for `Hashtable` (0.75). The default for initial capacity is currently unspecified (originally it was 101). Figure 14.9 shows the operations for the `HashSet` class. The `HashSet` class is not synchronized and permits null values.

The `HashMap` Class

The `HashMap` class implements the `Map` interface using a hash table. The `HashMap` class also uses a chaining method to resolve collisions. Like the `HashSet` class, the `HashMap` class is not synchronized and allows null values. Also like the previous implementations, the default load factor is 0.75. Like the `HashSet` class, the current default initial capacity is unspecified though it was also originally 101.

Figure 14.10 shows the operations on the `HashMap` class.

The `IdentityHashMap` Class

The `IdentityHashMap` class implements the `Map` interface using a hash table. The difference between this and the `HashMap` class is that the `IdentityHashMap` class

Return Value	Method	Description
	`HashSet()`	Constructs a new, empty set; the backing `HashMap` instance has the default capacity and load factor, which is 0.75.
	`HashSet(Collection c)`	Constructs a new set containing the elements in the specified collection.
	`HashSet(int initialCapacity)`	Constructs a new, empty set; the backing `HashMap` instance has the specified initial capacity and default load factor, which is 0.75.
	`HashSet(int initial Capacity, float loadFactor)`	Constructs a new, empty set; the backing `HashMap` instance has the specified initial capacity and the specified load factor.
`boolean`	`add(Object o)`	Adds the specified element to this set if it is not already present.
`void`	`clear()`	Removes all of the elements from this set.
`Object`	`clone()`	Returns a shallow copy of this `HashSet` instance: the elements themselves are not cloned.
`boolean`	`contains(Object o)`	Returns true if this set contains the specified element.
`boolean`	`isEmpty()`	Returns true if this set contains no elements.
`iterator()`	`iterator()`	Returns an iterator over the elements in this set.
`boolean`	`remove(Object o)`	Removes the given element from this set if it is present.
`int`	`size()`	Returns the number of elements in this set (its cardinality).

FIGURE 14.9 Operations on the `HashSet` class

uses reference-equality instead of object-equality when comparing both keys and values. This is the difference between using `key1==key2` and using `key1.equals(key2)`.

This class has one parameter: expected maximum size. This is the maximum number of key-value pairs that the table is expected to hold. If the table exceeds this maximum, then the table size will be increased and the table entries rehashed.

Figure 14.11 shows the operations on the `IdentityHashMap` class.

The `WeakHashMap` Class

The `WeakHashMap` class implements the `Map` interface using a hash table. This class is specifically designed with weak keys so that an entry in a `WeakHashMap` will automatically be removed when its key is no longer in use. In other words, if

Return Value	Method	Description
	`HashMap()`	Constructs a new, empty map with a default capacity and load factor, which is 0.75.
	`HashMap(int initial Capacity)`	Constructs a new, empty map with the specified initial capacity and default load factor, which is 0.75.
	`HashMap(int initial Capacity, float loadFactor)`	Constructs a new, empty map with the specified initial capacity and the specified load factor.
	`HashMap(Map t)`	Constructs a new map with the same mappings as the given map.
`void`	`clear()`	Removes all mappings from this map.
`Object`	`clone()`	Returns a shallow copy of this `HashMap` instance: the keys and values themselves are not cloned.
`boolean`	`containsKey(Object key)`	Returns true if this map contains a mapping for the specified key.
`boolean`	`containsValue (Object value)`	Returns true if this map maps one or more keys to the specified value.
`set`	`entrySet()`	Returns a collection view of the mappings contained in this map.
`Object`	`get(Object key)`	Returns the value to which this map maps the specified key.
`boolean`	`isEmpty()`	Returns true if this map contains no key-value mappings.
`Set`	`keySet()`	Returns a set view of the keys contained in this map.
`Object`	`put(Object key, Object value)`	Associates the specified value with the specified key in this map.
`void`	`putAll(Map t)`	Copies all of the mappings from the specified map to this one.
`Object`	`remove(Object key)`	Removes the mapping for this key from this map if present.
`int`	`size()`	Returns the number of key-value mappings in this map.
`Collection`	`values()`	Returns a collection view of the values contained in this map.

FIGURE 14.10 Operations on the `HashMap` class

the use of the key in a mapping in the `WeakHashMap` is the only remaining use of the key, the garbage collector will collect it anyway.

The `WeakHashMap` class allows both null values and null keys, and has the same tuning parameters as the `HashMap` class: initial capacity and load factor.

Figure 14.12 shows the operations on the `WeakHashMap` class.

Return Value	Method	Description
	`IdentityHashMap()`	Constructs a new, empty identity hash map with a default expected maximum size (21).
	`IdentityHashMap(int expectedMaxSize)`	Constructs a new, empty map with the specified expected maximum size.
	`IdentityHashMap(Map m)`	Constructs a new identity hash map containing the key-value mappings in the specified map.
`void`	`clear()`	Removes all mappings from this map.
`Object`	`clone()`	Returns a shallow copy of this identity hash map: the keys and values themselves are not cloned.
`boolean`	`containsKey(Object key)`	Tests whether the specified object reference is a key in this identity hash map.
`boolean`	`containsValue(Object value)`	Tests whether the specified object reference is a value in this identity hash map.
`Set`	`entrySet()`	Returns a set view of the mappings contained in this map.
`boolean`	`equals(Object o)`	Compares the specified object with this map for equality.
`Object`	`get(Object key)`	Returns the value to which the specified key is mapped in this identity hash map, or null if the map contains no mapping for this key.
`int`	`hashCode()`	Returns the hash code value for this map.
`boolean`	`isEmpty()`	Returns true if this identity hash map contains no key-value mappings.
`Set`	`keySet()`	Returns an identity-based set view of the keys contained in this map.
`Object`	`put(Object key, Object value)`	Associates the specified value with the specified key in this identity hash map.
`void`	`putAll(Map t)`	Copies all of the mappings from the specified map to this map. These mappings will replace any mappings that this map had for any of the keys currently in the specified map.
`Object`	`remove(Object key)`	Removes the mapping for this key from this map if present.
`int`	`size()`	Returns the number of key-value mappings in this identity hash map.
`Collection`	`values()`	Returns a collection view of the values contained in this map.

FIGURE 14.11 Operations on the `IdentityHashMap` class

Return Value	Method	Description
	`WeakHashMap()`	Constructs a new, empty `WeakHashMap` with the default initial capacity and the default load factor, which is 0.75.
	`WeakHashMap(int initialCapacity)`	Constructs a new, empty `WeakHashMap` with the given initial capacity and the default load factor, which is 0.75.
	`WeakHashMap(int initial Capacity, float loadFactor)`	Constructs a new, empty `WeakHashMap` with the given initial capacity and the given load factor.
	`WeakHashMap(Map t)`	Constructs a new `WeakHashMap` with the same mappings as the specified map.
`void`	`clear()`	Removes all mappings from this map.
`boolean`	`containsKey(Object key)`	Returns true if this map contains a mapping for the specified key.
`Set`	`entrySet()`	Returns a set view of the mappings in this map.
`Object`	`get(Object key)`	Returns the value to which this map maps the specified key.
`boolean`	`isEmpty()`	Returns true if this map contains no key-value mappings.
`Set`	`keySet()`	Returns a set view of the keys contained in this map.
`Object`	`put(Object key, Object value)`	Associates the specified value with the specified key in this map.
`void`	`putAll(Map t)`	Copies all of the mappings from the specified map to this map. These mappings will replace any mappings that this map had for any of the keys currently in the specified map.
`Object`	`remove(Object key)`	Removes the mapping for the given key from this map, if present.
`int`	`size()`	Returns the number of key-value mappings in this map.
`Collection`	`values()`	Returns a collection view of the values contained in this map.

FIGURE 14.12 Operations on the `WeakHashMap` class

`LinkedHashSet` and `LinkedHashMap`

The two remaining hashing implementations are extensions of previous classes. The `LinkedHashSet` class extends the `HashSet` class, and the `LinkedHashMap` class extends the `HashMap` class. Both of them are designed to solve the problem of iterator order. These implementations maintain a doubly linked list running through the entries to maintain the insertion order of the elements. Thus the iterator order for these implementations is the order in which the elements were inserted.

Figure 14.13 shows the additional operations for the `LinkedHashSet` class. Figure 14.14 shows the additional operations for the `LinkedHashMap` class.

Return Value	Method	Description
	`LinkedHashSet()`	Constructs a new, empty linked hash set with the default initial capacity (16) and load factor (0.75).
	`LinkedHashSet (Collection c)`	Constructs a new linked hash set with the same elements as the specified collection.
	`LinkedHashSet (int initialCapacity)`	Constructs a new, empty linked hash set with the specified initial capacity and the default load factor (0.75).
	`LinkedHashSet(int initialCapacity, float loadFactor)`	Constructs a new, empty linked hash set with the specified initial capacity and load factor.

FIGURE 14.13 Additional operations on the `LinkedHashSet` class

Return Value	Method	Description
	`LinkedHashMap()`	Constructs an empty insertion-ordered `LinkedHashMap` instance with a default capacity (16) and load factor (0.75).
	`LinkedHashMap (int initialCapacity)`	Constructs an empty insertion-ordered `LinkedHashMap` instance with the specified initial capacity and a default load factor (0.75).
	`LinkedHashMap (int initialCapacity, float loadFactor)`	Constructs an empty insertion-ordered `LinkedHashMap` instance with the specified initial capacity and load factor.
	`LinkedHashMap (int initialCapacity, float loadFactor, boolean accessOrder)`	Constructs an empty `LinkedHashMap` instance with the specified initial capacity, load factor, and ordering mode.
	`LinkedHashMap(Map m)`	Constructs an insertion-ordered `LinkedHashMap` instance with the same mappings as the specified map.
`void`	`clear()`	Removes all mappings from this map.
`boolean`	`containsValue (Object value)`	Returns true if this map maps one or more keys to the specified value.
`Object`	`get(Object key)`	Returns the value to which this map maps the specified key.
`protected boolean`	`removeEldestEntry (Map.Entry eldest)`	Returns true if this map should remove its eldest entry.

FIGURE 14.14 Additional operations on the `LinkedHashMap` class

Summary of Key Concepts

- In hashing, elements are stored in a hash table, with their location in the table determined by a hashing function.
- The situation where two elements or keys map to the same location in the table is called a collision.
- A hashing function that maps each element to a unique position in the table is said to be a perfect hashing function.
- Extraction involves using only a part of the element's value or key to compute the location at which to store the element.
- The division method is very effective when dealing with an unknown set of key values.
- In the shift folding method, the parts of the key are added together to create the index.
- The length-dependent method and the mid-square method may also be effectively used with strings by manipulating the binary representations of the characters in the string.
- Although Java provides a `hashcode` method for all objects, it is often preferable to define a specific hashing function for any particular class.
- The chaining method for handling collisions simply treats the hash table conceptually as a table of collections rather than as a table of individual cells.
- The open addressing method for handling collisions looks for another open position in the table other than the one to which the element is originally hashed.
- The load factor is the maximum percentage occupancy allowed in the hash table before it is resized.

Self-Review Questions

SR 14.1 What is the difference between a hash table and the other collections we have discussed?

SR 14.2 What is a collision in a hash table?

SR 14.3 What is a perfect hashing function?

SR 14.4 What is our goal for a hashing function?

SR 14.5 What is the consequence of not having a good hashing function?

SR 14.6 What is the extraction method?

SR 14.7 What is the division method?

SR 14.8 What is the shift folding method?

SR 14.9 What is the boundary folding method?

SR 14.10 What is the mid-square method?

SR 14.11 What is the radix transformation method?

SR 14.12 What is the digit analysis method?

SR 14.13 What is the length-dependent method?

SR 14.14 What is chaining?

SR 14.15 What is open addressing?

SR 14.16 What are linear probing, quadratic probing, and double hashing?

SR 14.17 Why is deletion from an open addressing implementation a problem?

SR 14.18 What is the load factor, and how does it affect table size?

Exercises

EX 14.1 Draw the hash table that results from adding the following integers (34 45 3 87 65 32 1 12 17) to a hash table of size 11 using the division method and linked chaining.

EX 14.2 Draw the hash table from Exercise 14.1 using a hash table of size 11 using array chaining with a total array size of 20.

EX 14.3 Draw the hash table from Exercise 14.1 using a table size of 17 and open addressing using linear probing.

EX 14.4 Draw the hash table from Exercise 14.1 using a table size of 17 and open addressing using quadratic probing.

EX 14.5 Draw the hash table from Exercise 14.1 using a table size of 17 and double hashing using extraction of the first digit as the secondary hashing function.

EX 14.6 Draw the hash table that results from adding the following integers (1983, 2312, 6543, 2134, 3498, 7654, 1234, 5678, 6789) to a hash table using shift folding of the first two digits with the last two digits. Use a table size of 13.

EX 14.7 Draw the hash table from Exercise 14.6 using boundary folding.

EX 14.8 Draw a UML diagram that shows how all of the various implementations of hashing within the Java Collections API are constructed.

Programming Projects

PP 14.1 Implement the hash table illustrated in Figure 14.1 using the array version of chaining.

PP 14.2 Implement the hash table illustrated in Figure 14.1 using the linked version of chaining.

PP 14.3 Implement the hash table illustrated in Figure 14.1 using open addressing with linear probing.

PP 14.4 Implement a dynamically resizable hash table to store people's names and Social Security numbers. Use the extraction method with division using the last four digits of the Social Security number. Use an initial table size of 31 and a load factor of 0.80. Use open addressing with double hashing using an extraction method on the first three digits of the Social Security number.

PP 14.5 Implement the problem from Programming Project 14.4 using linked chaining.

PP 14.6 Implement the problem from Programming Project 14.4 using the `HashMap` class of the Java Collections API.

PP 14.7 Create a new implementation of the bag collection called `HashtableBag` using a hash table.

PP 14.8 Implement the problem from Programming Project 14.4 using shift folding with the Social Security number divided into three equal three-digit parts.

PP 14.9 Create a graphical system that will allow a user to add and remove employees where each employee has an employee id (six-digit number), employee name, and years of service. Use the `hashcode` method of the `Integer` class as your hashing function and use one of the Java Collections API implementations of hashing.

PP 14.10 Complete Programming Project 14.9 using your own `hashcode` function. Use extraction of the first three digits of the employee id as the hashing function and use one of the Java Collections API implementations of hashing.

PP 14.11 Complete Programming Project 14.9 using your own `hashcode` function and your own implementation of a hash table.

PP 14.12 Create a system that will allow a user to add and remove vehicles from an inventory system. Vehicles will be represented by license number (an eight-character string), make, model, and color. Use your own array-based implementation of a hash table using chaining.

PP 14.13 Complete Programming Project 14.12 using a linked implementation with open addressing and double hashing.

Answers to Self-Review Questions

SRA 14.1 Elements are placed into a hash table at an index produced by a function of the value of the element or a key of the element. This is unique from other collections where the position/location of an element in the collection is determined either by comparison with the other values in the collection or by the order in which the elements were added or removed from the collection.

SRA 14.2 The situation where two elements or keys map to the same location in the table is called a collision.

SRA 14.3 A hashing function that maps each element to a unique position in the table is said to be a perfect hashing function.

SRA 14.4 We need a hashing function that will do a good job of distributing elements into positions in the table.

SRA 14.5 If we do not have a good hashing function, the result will be too many elements mapped to the same location in the table. This will result in poor performance.

SRA 14.6 Extraction involves using only a part of the element's value or key to compute the location at which to store the element.

SRA 14.7 The division method involves dividing the key by some positive integer p (usually the table size and usually prime) and then using the remainder as the index.

SRA 14.8 Shift folding involves dividing the key into parts (usually the same length as the desired index) and then adding the parts. Extraction or division is then used to get an index within the bounds of the table.

SRA 14.9 Like shift folding, boundary folding involves dividing the key into parts (usually the same length as the desired index). However, some of the parts are then reversed before adding. One example is to imagine that the parts are written side by side on a piece of paper, which is then folded on the boundaries between parts. In this way, every other part is reversed.

SRA 14.10 The mid-square method involves multiplying the key by itself and then extracting some number of digits or bytes from the middle of the result. Division can then be used to guarantee an index within the bounds of the table.

SRA 14.11 The radix transformation method is a variation on the division method where the key is first converted to another numeric base and then divided by the table size with the remainder used as the index.

SRA 14.12 In the digit analysis method, the index is formed by extracting, and then manipulating, specific digits from the key.

SRA 14.13 In the length-dependent method, the key and the length of the key are combined in some way to form either the index itself or an intermediate value that is then used with one of our other methods to form the index.

SRA 14.14 The chaining method for handling collisions simply treats the hash table conceptually as a table of collections rather than as a table of individual cells. Thus each cell is a pointer to the collection associated with that location in the table. Usually this internal collection is either an unordered list or an ordered list.

SRA 14.15 The open addressing method for handling collisions looks for another open position in the table other than the one to which the element is originally hashed.

SRA 14.16 Linear probing, quadratic probing, and double hashing are methods for determining the next table position to try if the original hash causes a collision.

SRA 14.17 Because of the way that a path is formed in open addressing, deleting an element from the middle of that path can cause elements beyond that on the path to be unreachable.

SRA 14.18 The load factor is the maximum percentage occupancy allowed in the hash table before it is resized. Once the load factor has been reached, a new table is created that is twice the size of the current table, and then all of the elements in the current table are inserted into the new table.