2. For the `Map index` in Example 7.4, what key–value pairs would be stored for each token in the following data file?

```
this line is first
and line 2 is second
followed by the third line
```

3. Explain the effect of each statement in the following fragment on the index built in Self-Check Exercise 2.

```
lines = index.get("this");
lines = index.get("that");
lines = index.get("line");
lines.add(4);
index.put("is", lines);
```

PROGRAMMING

1. Write statements to create a `Map` object that will store each word occurring in a term paper along with the number of times the word occurs.

2. Write a method `buildWordCounts` (based on `buildIndex`) that builds the `Map` object described in Programming Exercise 1.

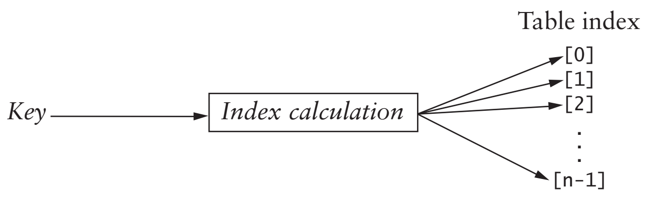
7.3 Hash Tables

Before we discuss the details of implementing the required methods of the `Set` and `Map` interfaces, we will describe a data structure, the *hash table*, that can be used as the basis for such an implementation. The goal behind the hash table is to be able to access an entry based on its key value, not its location. In other words, we want to be able to access an element directly through its key value rather than having to determine its location first by searching for the key value in an array. (This is why the `Set` interface has method `contains(obj)` instead of `get(index)`.) Using a hash table enables us to retrieve an item in constant time (expected **O**(1)). We say expected **O**(1) rather than just **O**(1) because there will be some cases where the performance will be much worse than **O**(1) and may even be **O**(*n*), but on the average, we expect that it will be **O**(1). Contrast this with the time required for a linear search of an array, **O**(*n*), and the time to access an element in a binary search tree, **O**(log *n*).

Hash Codes and Index Calculation

The basis of hashing (and hash tables) is to transform the item's key value to an integer value (its *hash code*) that will then be transformed into a table index. Figure 7.4 illustrates this process for a table of size *n*. We discuss how this might be done in the next few examples.

FIGURE 7.4
Index Calculation for a Key



EXAMPLE 7.5 Consider the Huffman code problem discussed in Section 6.7. To build the Huffman tree, you needed to know the number of occurrences of each character in the text being encoded. Let's assume that the text contained only the ASCII characters (the first 128 Unicode values starting with `\u0000`). We could use a table of size 128, one element for each possible character, and let the Unicode for each character be its location in the table. Using this approach, table element 65 would give us the number of occurrences of the letter A, table element 66 would give us the number of occurrences of the letter B, and so on. The hash code for each character is its Unicode value (a number), which is also its index in the table. In this case, we could calculate the table index for character `asciiChar` using the following assignment statement, where `asciiChar` represents the character we are seeking in the table:

```
int index = asciiChar;
```

EXAMPLE 7.6 Let's consider a slightly harder problem: assume that any of the Unicode characters can occur in the text, and we want to know the number of occurrences of each character. There are over 65,000 Unicode characters, however. For any file, let's assume that at most 100 different characters actually appear. So, rather than use a table with 65,536 elements, it would make sense to try to store these items in a much smaller table (say, 200 elements). If the hash code for each character is its Unicode value, we need to convert this value (between 0 and 65,536) to an array index between 0 and 199. We can calculate the array index for character `uniChar` as

```
int index = uniChar % 200
```

Because the range of Unicode values (the key range) is much larger than the `index` range, it is likely that some characters in our text will have the same index value. Because we can store only one key–value pair in a given array element, a situation known as a *collision* results. We discuss how to deal with collisions shortly.

Methods for Generating Hash Codes

In most applications, the keys that we will want to store in a table will consist of strings of letters or digits rather than a single character (e.g., a social security number, a person's name, or a part ID). We need a way to map each string to a particular table index. Again, we have a situation in which the number of possible key values is much larger than the table size. For example, if a string can store up to 10 letters or digits, the number of possible strings is 36^{10} (approximately 3.7×10^{15}), assuming the English alphabet with 26 letters.

Generating good hash codes for arbitrary strings or arbitrary objects is somewhat of an experimental process. Simple algorithms tend to generate a lot of collisions. For example, simply summing the `int` values for all characters in a string would generate the same hash code for words that contained the same letters but in different orders, such as "sign" and "sing", which would have the same hash code using this algorithm (`'s' + 'i' + 'n' + 'g'`). The algorithm used by the Java API accounts for the position of the characters in the string as well as the character values.

The `String.hashCode()` method returns the integer calculated by the formula:

$$s_0 \times 31^{(n-1)} + s_1 \times 31^{(n-2)} + \cdots + s_{n-1}$$

where s_i is the *i*th character of the string and `n` is the length of the string. For example, the string `"Cat"` would have a hash code of `'C'` × 31² + `'a'` × 31 + `'t'`. This is the number 67,510. (The number 31 is a prime number that generates relatively few collisions.)

As previously discussed, the integer value returned by method `String.hashCode` can't be unique because there are too many possible strings. However, the probability of two strings having the same hash code value is relatively small because the `String.hashCode` method distributes the hash code values fairly evenly throughout the range of **`int`** values.

Because the hash codes are distributed evenly throughout the range of **`int`** values, method `String.hashCode` will appear to produce a random value, as will the expressions `s.hashCode() % table.length`, which selects the initial value of `index` for `String s`. If the object is not already present in the table, the probability that this expression does not yield an empty slot in the table is proportional to how full the table is.

One additional criterion for a good hash function, besides a random distribution for its values, is that it be relatively simple and efficient to compute. It doesn't make much sense to use a hash function whose computation is an **O**(n) process to avoid doing an **O**(n) search.

Open Addressing

Next, we consider two ways to organize hash tables: open addressing and chaining. In open addressing, each hash table element (type `Object`) references a single key–value pair. We can use the following simple approach (called *linear probing*) to access an item in a hash table. If the index calculated for an item's key is occupied by an item with that key, we have found the item. If that element contains an item with a different key, we increment the index by 1. We keep incrementing the index (modulo the table length) until either we find the key we are seeking or we reach a **`null`** entry. A **`null`** entry indicates that the key is not in the table.

Algorithm for Accessing an Item in a Hash Table

```
1.  Compute the index by taking the item's hashCode() % table.length.
2.  if table[index] is null
3.      The item is not in the table.
4.  else if table[index] is equal to the item
5.      The item is in the table.
    else
6.      Continue to search the table by incrementing the index until either the item is found
        or a null entry is found.
```

Step 1 ensures that the `index` is within the table range (0 through `table.length – 1`). If the condition in Step 2 is true, the table index does not reference an object, so the item is not in the table. The condition in Step 4 is true if the item being sought is at position `index`, in which case the item is located. Steps 1 through 5 can be done in **O**(1) expected time.

Step 6 is necessary for two reasons. The values returned by method `hashCode` are not unique, so the item being sought can have the same hash code as another one in the table. Also, the remainder calculated in Step 1 can yield the same index for different hash code values. Both of these cases are examples of collisions.

Table Wraparound and Search Termination

Note that as you increment the table index, your table should wrap around (as in a circular array) so that the element with subscript 0 "follows" the element with subscript `table.length - 1`. This enables you to use the entire table, not just the part with subscripts larger than the hash code value, but it leads to the potential for an infinite loop in Step 6 of the algorithm. If the table is full and the objects examined so far do not match the one you are seeking, how do you know when to stop? One approach would be to stop when the index value for the next probe is the same as the hash code value for the object. This means that you have come full circle to the starting value for the index. A second approach would be to ensure that the table is never full by increasing its size after an insertion if its occupancy rate exceeds a specified threshold. This is the approach that we take in our implementation.

EXAMPLE 7.7 We illustrate insertion of five names in a table of size 5 and in a table of size 11. Table 7.3 shows the names, the corresponding hash code, the hash code modulo 5 (in column 3), and the hash code modulo 11 (in column 4). We picked prime numbers (5 and 11) because empirical tests have shown that hash tables with a size that is a prime number often give better results.

For a table of size 5 (an occupancy rate of 100 percent), `"Tom"`, `"Dick"`, and `"Sam"` have hash indexes of 4, and `"Harry"` and `"Pete"` have hash indexes of 3; for a table length of 11 (an occupancy rate of 45 percent), `"Dick"` and `"Sam"` have hash indexes of 5, but the others have hash indexes that are unique. We see how the insertion process works next.

For a table of size 5, if `"Tom"` and `"Dick"` are the first two entries, `"Tom"` would be stored at the element with index 4, the last element in the table. Consequently, when `"Dick"` is inserted, because element 4 is already occupied, the hash index is incremented to 0 (the table wraps around to the beginning), where `"Dick"` is stored.

TABLE 7.3
Names and `hashCode` Values for Table Sizes 5 and 11

Name	hashCode()	hashCode()%5	hashCode()%11
`"Tom"`	84274	4	3
`"Dick"`	2129869	4	5
`"Harry"`	69496448	3	10
`"Sam"`	82879	4	5
`"Pete"`	2484038	3	7

[0]	`"Dick"`
[1]	**`null`**
[2]	**`null`**
[3]	**`null`**
[4]	`"Tom"`

`"Harry"` is stored in position 3 (the hash index), and `"Sam"` is stored in position 1 because its hash index is 4 but the elements at 4 and 0 are already filled.

[0]	`"Dick"`
[1]	`"Sam"`
[2]	**`null`**
[3]	`"Harry"`
[4]	`"Tom"`

Finally, `"Pete"` is stored in position 2 because its hash index is 3 but the elements at positions 3, 4, 0, 1 are filled.

[0]	`"Dick"`
[1]	`"Sam"`
[2]	`"Pete"`
[3]	`"Harry"`
[4]	`"Tom"`

For the table of size 11, the entries would be stored as shown in the following table, assuming that they were inserted in the order `"Tom"`, `"Dick"`, `"Harry"`, `"Sam"`, and finally `"Pete"`. Insertions go more smoothly for the table of size 11. The first collision occurs when `"Sam"` is stored, so `"Sam"` is stored at position 6 instead of position 5.

[0]	**`null`**
[1]	**`null`**
[2]	**`null`**
[3]	`"Tom"`
[4]	**`null`**
[5]	`"Dick"`
[6]	`"Sam"`
[7]	`"Pete"`
[8]	**`null`**
[9]	**`null`**
[10]	`"Harry"`

For the table of size 5, retrieval of `"Tom"` can be done in one step. Retrieval of all of the others would require a linear search because of collisions that occurred when they were inserted. For the table of size 11, retrieval of all but `"Sam"` can be done in one step, and retrieval of `"Sam"` requires only two steps. This example illustrates that the best way to reduce the probability of a collision is to increase the table size.

Traversing a Hash Table

One thing that you cannot do is traverse a hash table in a meaningful way. If you visit the hash table elements in sequence and display the objects stored, you would display the strings `"Dick"`, `"Sam"`, `"Pete"`, `"Harry"`, and `"Tom"` for the table of length 5 and the strings `"Tom"`, `"Dick"`, `"Sam"`, `"Pete"`, and `"Harry"` for a table of length 11. In either case, the list of names is in arbitrary order.

Deleting an Item Using Open Addressing

When an item is deleted, we cannot just set its table entry to **`null`**. If we do, then when we search for an item that may have collided with the deleted item, we may incorrectly conclude that the item is not in the table. (Because the item that collided was inserted after the deleted item, we will have stopped our search prematurely.) By storing a dummy value when an item is deleted, we force the search algorithm to keep looking until either the desired item is found or a **`null`** value, representing a free cell, is located.

Although the use of a dummy value solves the problem, keep in mind that it can lead to search inefficiency, particularly when there are many deletions. Removing items from the table does not reduce the search time because the dummy value is still in the table and is part of a search chain. In fact, you cannot even replace a deleted value with a new item because you still need to go to the end of the search chain to ensure that the new item is not already present in the table. So deleted items waste storage space and reduce search efficiency. In the worst case, if the table is almost full and then most of the items are deleted, you will have $O(n)$ performance when searching for the few items remaining in the table.

Reducing Collisions by Expanding the Table Size

Even with a good hashing function, it is still possible to have collisions. The first step in reducing these collisions is to use a prime number for the size of the table.

In addition, the probability of a collision is proportional to how full the table is. Therefore, when the hash table becomes sufficiently full, a larger table should be allocated and the entries reinserted.

We previously saw examples of expanding the size of an array. Generally, what we did was to allocate a new array with twice the capacity of the original, copy the values in the original array to the new array, and then reference the new array instead of the original. This approach will not work with hash tables. If you use it, some search chains will be broken because the new table does not wrap around in the same way as the original table. The last element in the original table will be in the middle of the new table, and it does not wrap around to the first element of the new table. Therefore, you expand a hash table (called *rehashing*) using the following algorithm.

Algorithm for Rehashing

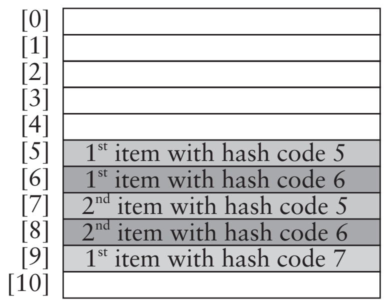
1. Allocate a new hash table with twice the capacity of the original.
2. Reinsert each old table entry that has not been deleted into the new hash table.
3. Reference the new table instead of the original.

Step 2 reinserts each item from the old table into the new table instead of copying it over to the same location. We illustrate this in the hash table implementation. Note that deleted items are not reinserted into the new table, thereby saving space and reducing the length of some search chains.

Reducing Collisions Using Quadratic Probing

The problem with linear probing is that it tends to form clusters of keys in the table, causing longer search chains. For example, if the table already has keys with hash codes of 5 and 6, a new item that collides with either of these keys will be placed at index 7. An item that collides with any of these three items will be placed at index 8, and so on. Figure 7.5 shows a hash table of size 11 after inserting elements with hash codes in the sequence 5, 6, 5, 6, 7. Each new collision expands the cluster by one element, thereby increasing the length of the search chain for each element in that cluster. For example, if another element is inserted with any hash code in the range 5 through 9, it will be placed at position 10, and the search chain for items with hash codes of 5 and 6 would include the elements at indexes 7, 8, 9, and 10.

FIGURE 7.5
Clustering with Linear Probing


[0]
[1]
[2]
[3]
[4]
[5] 1st item with hash code 5
[6] 1st item with hash code 6
[7] 2nd item with hash code 5
[8] 2nd item with hash code 6
[9] 1st item with hash code 7
[10]

One approach to reduce the effect of clustering is to use *quadratic probing* instead of linear probing. In quadratic probing, the increments form a quadratic series $(1 + 2^2 + 3^2 + \cdots)$. Therefore, the next value of `index` is calculated using the steps:

```
probeNum++;
index = (startIndex + probeNum * probeNum) % table.length
```

where `startIndex` is the index calculated using method `hashCode` and `probeNum` starts at 0. Ignoring wraparound, if an item has a hash code of 5, successive values of `index` will be 6 (5 + 1), 9 (5 + 4), 14 (5 + 9), . . ., instead of 6, 7, 8, ... Similarly, if the hash code is 6, successive values of `index` will be 7, 10, 15, and so on. Unlike linear probing, these two search chains have only one table element in common (at index 6).

Figure 7.6 illustrates the hash table after elements with hash codes in the same sequence as in the preceding table (5, 6, 5, 6, 7) have been inserted with quadratic probing. Although the cluster of elements looks similar, their search chains do not overlap as much as before. Now the search chain for an item with a hash code of 5 consists of the elements at 5, 6, and 9, and the search chain for an item with a hash code of 6 consists of the elements at positions 6 and 7.

Problems with Quadratic Probing

One disadvantage of quadratic probing is that the next index calculation is a bit time-consuming as it involves a multiplication, an addition, and a modulo division. A more efficient way to calculate the next index follows:

```
k += 2;
index = (index + k) % table.length;
```

which replaces the multiplication with an addition. If the initial value of `k` is –1, successive values of `k` will be 1, 3, 5, 7, ... If the hash code is 5, successive values of `index` will be 5, 6 (5 + 1), 9 (5 + 1 + 3), 14 (5 + 1 + 3 + 5),. . . . The proof of the equality of these two approaches to calculating `index` is based on the following mathematical series:

$$n^2 = 1 + 3 + 5 + \cdots + 2n - 1$$

FIGURE 7.6
Insertion with Quadratic Probing

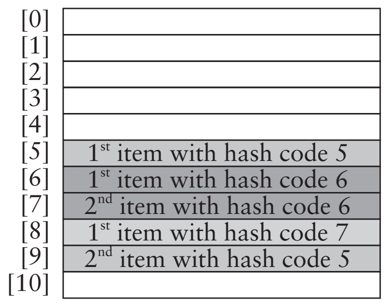
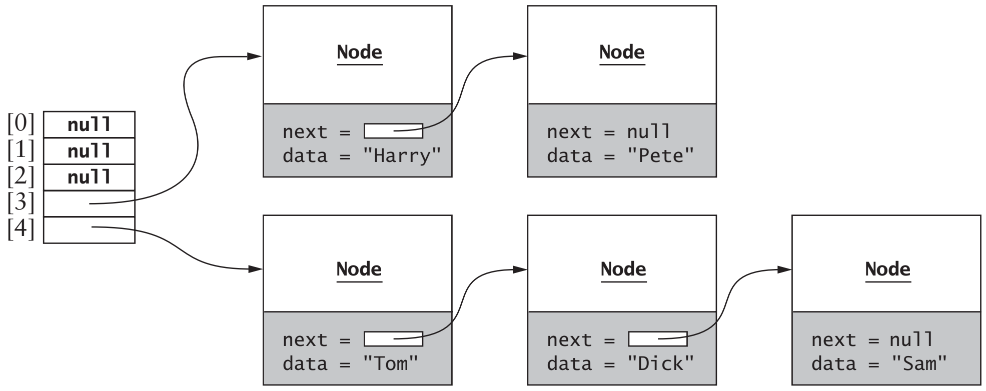

[0]
[1]
[2]
[3]
[4]
[5] 1st item with hash code 5
[6] 1st item with hash code 6
[7] 2nd item with hash code 6
[8] 1st item with hash code 7
[9] 2nd item with hash code 5
[10]

FIGURE 7.7
Example of Chaining



A more serious problem with quadratic probing is that not all table elements are examined when looking for an insertion index, so it is possible that an item can't be inserted even when the table is not full. It is also possible that your program can get stuck in an infinite loop while searching for an empty slot. It can be proved that if the table size is a prime number and the table is never more than half full, this can't happen. However, requiring that the table be half empty at all times wastes quite a bit of memory. For these reasons, we will use linear probing in our implementation.

Chaining

An alternative to open addressing is a technique called *chaining,* in which each table element references a linked list that contains all the items that hash to the same table index. This linked list is often called a *bucket,* and this approach is sometimes called *bucket hashing*. Figure 7.7 shows the result of chaining for our earlier example with a table of size 5. Each new element with a particular hash index can be placed at the beginning or the end of the associated linked list. The algorithm for accessing such a table is the same as for open addressing, except for the step for resolving collisions. Instead of incrementing the table index to access the next item with a particular hash code value, you traverse the linked list referenced by the table element with index `hashCode() % table.length`.

One advantage of chaining is that only items that have the same value for `hashCode() % table.length` will be examined when looking for an object. In open addressing, search chains can overlap, so a search chain may include items in the table that have different starting index values.

A second advantage is that you can store more elements in the table than the number of table slots (indexes), which is not the case for open addressing. If each table index already references a linked list, additional items can be inserted in an existing list without increasing the table size (number of indexes).

Once you have determined that an item is not present, you can insert it either at the beginning or at the end of the list. To delete an item, simply remove it from the list. In contrast to open addressing, removing an item actually deletes it, so it will not be part of future search chains.

Performance of Hash Tables

The *load factor* for a hash table is the number of filled cells divided by table size. The load factor has the greatest effect on hash table performance. The lower the load factor, the better the performance because there is less chance of a collision when a table is sparsely populated. If there are no collisions, the performance for search and retrieval is **O**(1), regardless of the table size.

Performance of Open Addressing Versus Chaining

Donald E. Knuth (*Searching and Sorting*, vol. 3 of *The Art of Computer Programming*, Addison-Wesley, 1973) derived the following formula for the expected number of comparisons, c, required for finding an item that is in a hash table using open addressing with linear probing and a load factor L:

$$c = \frac{1}{2}\left(1 + \frac{1}{1-L}\right)$$

TABLE 7.4
Number of Probes for Different Values of Load Factor (L)

L	Number of Probes with Linear Probing	Number of Probes with Chaining
0.0	1.00	1.00
0.25	1.17	1.13
0.5	1.50	1.25
0.75	2.50	1.38
0.85	3.83	1.43
0.9	5.50	1.45
0.95	10.50	1.48

Table 7.4 (second column) shows the value of c for different values of load factor (L). It shows that if L is 0.5 (half full), the expected number of comparisons required is 1.5. If L increases to 0.75, the expected number of comparisons is 2.5, which is still very respectable. If L increases to 0.9 (90 percent full), the expected number of comparisons is 5.5. This is true regardless of the size of the table.

Using chaining, if an item is in the table, on average we have to examine the table element corresponding to the item's hash code and then half of the items in each list. The average number of items in a list is L, the number of items divided by the table size. Therefore, we get the formula

$$c = 1 + \frac{L}{2}$$

for a successful search. Table 7.4 (third column) shows the results for chaining. For values of L between 0.0 and 0.75, the results are similar to those of linear probing, but chaining gives better performance than linear probing for higher load factors. Quadratic probing (not shown) gives performance that is between those of linear probing and chaining.

Performance of Hash Tables versus Sorted Arrays and Binary Search Trees

If we compare hash table performance with binary search of a sorted array, the number of comparisons required by binary search is **O**(log n), so the number of comparisons increases with table size. A sorted array of size 128 would require up to 7 probes (2^7 is 128), which is more than for a hash table of any size that is 90 percent full. A sorted array of size 1024 would require up to 10 probes (2^{10} is 1024). A binary search tree would yield the same results.

You can insert into or remove elements from a hash table in **O**(1) expected time. Insertion or removal from a binary search tree is **O**(log n), but insertion or removal from a sorted array is **O**(n) (you need to shift the larger elements over). (Worst-case performance for a hash table or a binary search tree is **O**(n).)

Storage Requirements for Hash Tables, Sorted Arrays, and Trees

The performance of hashing is certainly preferable to that of binary search of an array (or a binary search tree), particularly if L is less than 0.75. However, the tradeoff is that the lower the load factor, the more unfilled storage cells there are in a hash table, whereas there are no empty cells in a sorted array. Because a binary search tree requires three references per node (the item, the left subtree, and the right subtrees), more storage would be required for a binary search tree than for a hash table with a load factor of 0.75.

EXAMPLE 7.8 A hash table of size 100 with open addressing could store 75 items with a load factor of 0.75. This would require storage for 100 references. This would require storage for 100 references (25 references would be `null`).

Storage Requirements for Open Addressing and Chaining

Next, we consider the effect of chaining on storage requirements. For a table with a load factor of L, the number of table elements required is n (the size of the table). For open addressing, the number of references to an item (a key–value pair) is n. For chaining, the average number of nodes in a list is L. If we use the Java API `LinkedList`, there will be three references in each node (the item, the next list element, and the previous element). However, we could use our own single-linked list and eliminate the previous-element reference (at some time cost for deletions). Therefore, we will require storage for $n + 2L$ references.

EXAMPLE 7.9 If we have 60,000 items in our hash table and use open addressing, we would need a table size of 80,000 to have a load factor of 0.75 and an expected number of comparisons of 2.5. Next, we calculate the table size, n, needed to get similar performance using chaining.

$$\begin{aligned} 2.5 &= 1 + \frac{L}{2} \\ 5.0 &= 2 + L \\ 3.0 &= \frac{60{,}000}{n} \\ n &= 20{,}000 \end{aligned}$$

A hash table of size 20,000 requires storage space for 20,000 references to lists. There will be 60,000 nodes in the table (one for each item). If we use linked lists of nodes, we will need storage for 140,000 references (2 references per node plus the 20,000 table references). This is almost twice the storage needed for open addressing.

EXERCISES FOR SECTION 7.3

SELF-CHECK

1. For the hash table search algorithm shown in this section, why was it unnecessary to test whether all table entries had been examined as part of Step 5?
2. For the items in the five-element table of Table 7.3, compute `hashCode() % table.length` for lengths of 7 and 13. What would be the position of each word in tables of

these sizes using open addressing and linear probing? Answer the same question for chaining.

3. The following table stores `Integer` keys with the **`int`** values shown. Show one sequence of insertions that would store the keys as shown. Which elements were placed in their current position because of collisions? Show the table that would be formed by chaining.

Index	Key
[0]	24
[1]	6
[2]	20
[3]	
[4]	14

4. For Table 7.3 and the table size of 5 shown in Example 7.7, discuss the effect of deleting the entry for Dick and replacing it with a **`null`** value. How would this affect the search for Sam, Pete, and Harry? Answer both questions if you replace the entry for Dick with the string `"deleted"` instead of **`null`**.

5. Explain what is wrong with the following strategy to reclaim space that is filled with deleted items in a hash table: when attempting to insert a new item in the table, if you encounter an item that has been deleted, replace the deleted item with the new item.

6. Compare the storage requirement for a hash table with open addressing, a table size of 500, and a load factor of 0.5 with a hash table that uses chaining and gives the same performance.

7. One simple hash code is to use the sum of the ASCII codes for the letters in a word. Explain why this is not a good hash code.

8. If p_i is the position of a character in a string and c_i is the code for that character, would $c_1p_1 + c_2p_2 + c_3p_3 + \ldots$ be a better hash code? Explain why or why not.

9. Use the hash code in Self-Check Exercise 7 to store the words `"cat"`, `"hat"`, `"tac"`, and `"act"` in a hash table of size 10. Show this table using open hashing and chaining.

PROGRAMMING

1. Code the following algorithm for finding the location of an object as a static method. Assume a hash table array and an object to be located in the table are passed as arguments. Return the object's position if it is found; return –1 if the object is not found.

```
1.   Compute the index by taking the hashCode() % table.length.
2.   if table[index] is null
3.       The object is not in the table.
     else if table[index] is equal to the object
4.       The object is in the table.
     else
5.       Continue to search the table (by incrementing index) until either the
         object is found or a null entry is found.
```