CHAPTER 11

Hash Tables

The search problem, which was introduced in Chapter 4, attempts to locate an item in a collection based on its associated search key. Searching is the most common operation applied to collections of data. It's not only used to determine if an item is in the collection, but can also be used in adding new items to the collection and removing existing items. Given the importance of searching, we need to be able to accomplish this operation fast and efficiently.

If the collection is stored in a sequence, we can use a linear search to locate an item. The linear search is simple to implement but not very efficient as it requires $O(n)$ time in the worst case. We saw that the search time could be improved using the binary search algorithm as it only requires $O(\log n)$ time in the worst case. But the binary search can only be applied to a sequence in which the keys are in sorted order.

The question becomes, can we improve the search operation to achieve better than $O(\log n)$ time? The linear and binary search algorithms are both ***comparison-based searches***. That is, in order to locate an item, the target search key has to be compared against the other keys in the collection. Unfortunately, it can be shown that $O(\log n)$ is the best we can achieve for a comparison-based search. To improve on this time, we would have to use a technique other than comparing the target key against other keys in the collection. In this chapter, we explore the use of a non-comparison-based algorithm to provide a more efficient search operation. This is the same technique used in the implementation of Python's dictionary structure.

11.1 Introduction

Suppose you have a collection of products for which you need to maintain information and allow for numerous searches on that information. At the present time, you only have a small collection of products but you can envision having up to

a hundred products in the future. So, you decide to assign a unique identifier or code to each product using the integer values in the range 100 . . . 199. To manage the data and allow for searches, you decide to store the product codes in an array of sufficient size for the number of products available.

Figure 11.1 illustrates the contents of the array for a collection of nine product codes. Depending on the number of searches, we can choose whether to perform a simple linear search on the array or first sort the keys and then use a binary search. Even though this example uses a small collection, in either case the searches still require at least logarithmic time and possibly even linear time in the worst case.

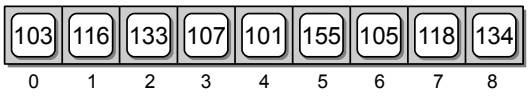

Figure 11.1: A collection of product codes stored in an array.

Given the small range of key values, this problem is a special case. The searches can actually be performed in constant time. Instead of creating an array that is only large enough to hold the products on hand, suppose we create an array with 100 elements, the size needed to store all possible product codes. We can then assign each key a specific element in the array. If the product code exists, the key and its associated data will be stored in its assigned element. Otherwise, the element will be set to `None` to flag the absence of that product. The search operation is reduced to simply examining the array element associated with a given search key to determine if it contains a valid key or a null reference.

To determine the element assigned to a given key, we note that the product codes are in the range [100 . . . 199] while the array indices are in the range [0 . . . 99]. There is a natural mapping between the two. Key 100 can be assigned to element 0, key 101 to element 1, key 102 to element 2, and so on. This mapping can be computed easily by subtracting 100 from the key value or with the use of the modulus operator (`key % 100`). Figure 11.2 illustrates the storage of our sample product collection using this approach.

This technique provides ***direct access*** to the search keys. When searching for a key, we apply the same mapping operation to determine the array element that contains the given target. For example, suppose we want to search for product 107. We compute `107 % 100` to determine the key will be in element 7 if it exists. Since

NOTE

Search Keys. Throughout the text, we have focused on the storage and use of search keys when discussing the search problem. But remember, the search keys are commonly associated with a data record and are used as the unique identifier for that record. While our examples have only illustrated the keys, we assume the associated data is also stored along with the search key.

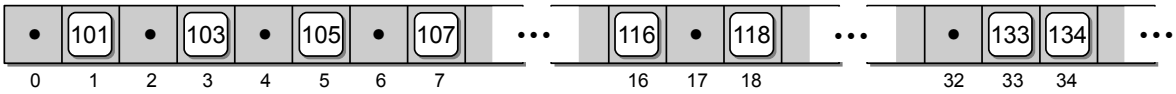

Figure 11.2: Storing a collection of product codes by direct mapping.

there is a product with code 107 and it can be directly accessed at array element 7. If the target key is not in the collection, as is the case for product code 102, the corresponding element (`102 % 100 = 2`) will contain a null reference. This results in a constant time search since we can directly examine a specific element of the array and not have to compare the target key against the other keys in the collection.

11.2 Hashing

We can use the direct access technique for small sets of keys that are composed of consecutive integer values. But what if the key can be any integer value? Even with a small collection of keys, we cannot create an array large enough to store all possible integer values. That's where hashing comes into play.

Hashing is the process of mapping a search key to a limited range of array indices with the goal of providing direct access to the keys. The keys are stored in an array called a ***hash table*** and a ***hash function*** is associated with the table. The function converts or maps the search keys to specific entries in the table. For example, suppose we have the following set of keys:

```
765, 431, 96, 142, 579, 226, 903, 388
```

and a hash table, T, containing $M = 13$ elements. We can define a simple hash function $h(\cdot)$ that maps the keys to entries in the hash table:

$$h(\text{key}) = \text{key } \texttt{\%}\ M$$

You will notice this is the same operation we used with the product codes in our earlier example. Dividing the integer key by the size of the table and taking the remainder ensures the value returned by the function will be within the valid range of indices for the given table.

To add keys to the hash table, we apply the hash function to determine the entry in which the given key should be stored. Applying the hash function to key 765 yields a result of 11, which indicates 765 should be stored in element 11 of the hash table. Likewise, if we apply the hash function to the next four keys in the list, we find:

```
h(431) => 2    h(96)  => 5    h(142) => 12    h(579) => 7
```

all of which are unique index values. Figure 11.3 illustrates the insertion of the first five keys into the hash table.

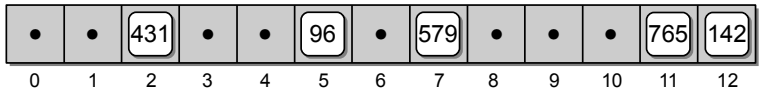

Figure 11.3: Storing the first five keys in the hash table.

11.2.1 Linear Probing

The first five keys were easily added to the table. The resulting index values were unique and the corresponding table entries contained null references, which indicated empty slots. But that's not always the case. Consider what happens when we attempt to add key 226 to the hash table. The hash function maps this key to entry 5, but that entry already contains key 96, as illustrated in Figure 11.4. The result is a ***collision***, which occurs when two or more keys map to the same hash location. We mentioned earlier that the goal of hashing is to provide direct access to a collection of search keys. When the key value can be one of a wide range of values, it's impossible to provide a unique entry for all possible key values.

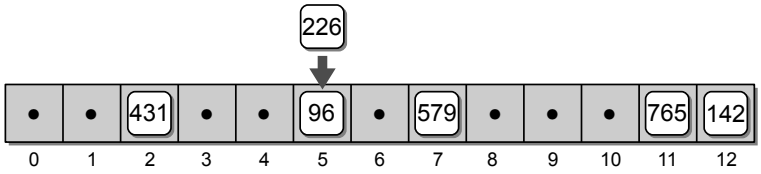

Figure 11.4: A collision occurs when adding key 226.

If two keys map to the same table entry, we must resolve the collision by ***probing*** the table to find another available slot. The simplest approach is to use a ***linear probe***, which examines the table entries in sequential order starting with the first entry immediately following the original hash location. For key value 226, the linear probe finds slot 6 available, so the key can be stored at that position, as illustrated in Figure 11.5.

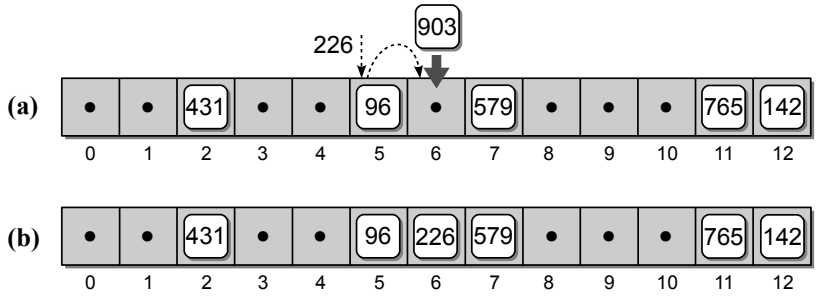

Figure 11.5: Resolving a collision for key 226 requires adding the key to the next slot.

When key 903 is added, the hash function maps the key to index 6, but we just added key 226 to this entry. Your first instinct may be to remove key 226 from this location, since 226 did not map directly to this entry, and store 903 here instead. Once a key is stored in the hash table, however, it's only removed when a delete operation is performed. This collision has to be resolved just like any other, by probing to find another slot. In the case of key 903, the linear probe leads us to slot 8, as illustrated in Figure 11.6.

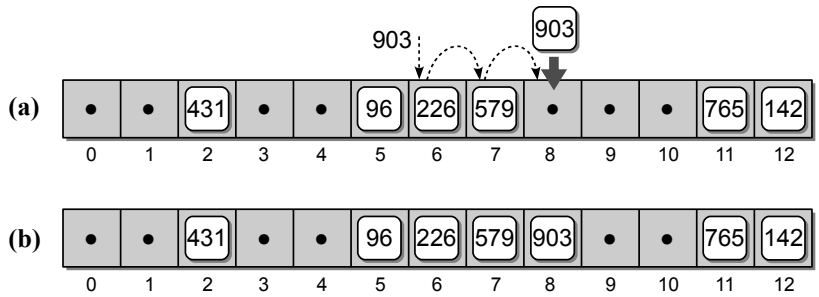

Figure 11.6: Adding key 903 to the hash table: (a) performing a linear probe; and (b) the result after adding the key.

If the end of the array is reached during the probe, we have to wrap around to the first entry and continue until either an available slot is found or all entries have been examined. For example, if we add key 388 to the hash table, the hash function maps the key to slot 11, which contains key 765. The linear probe, as illustrated in Figure 11.7, requires wrapping around to the beginning of the array.

Searching

Searching a hash table for a specific key is very similar to the add operation. The target key is mapped to an initial slot in the table and then it is determined if that entry contains the key. If the key is not at that location, the same probe used to add the keys to the table must be used to locate the target. In this case, the

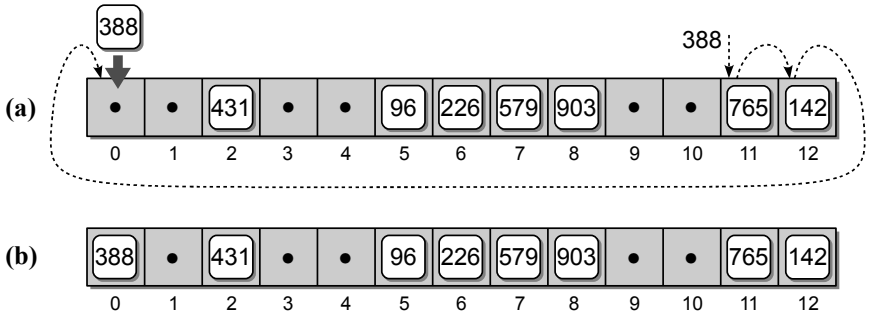

Figure 11.7: Adding key 388 to the hash table: (a) performing a linear probe; and (b) the result after adding the key.

probe continues until the target is located, a null reference is encountered, or all slots have been examined. When either of the latter two situations occurs, this indicates the target key is not in the table. Figure 11.8 illustrates the searches for key 903, which is in the table, and key 561, which is not in the table.

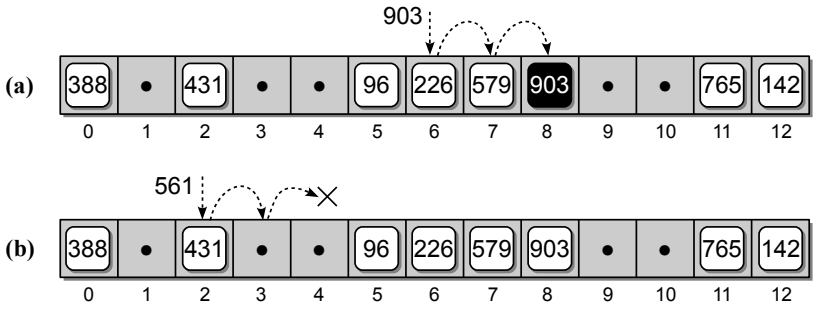

Figure 11.8: Searching the hash table: (a) a successful search for key 903 and (b) an unsuccessful search for key 561.

Deletions

We've seen how keys are added to the table with the use of the hash function and a linear probe for resolving collisions. But how are deletions handled? Deleting from a hash table is a bit more complicated than an insertion. A search can be performed to locate the key in a similar fashion as the basic search operation described earlier. But after finding the key, we cannot simply remove it by setting the corresponding table entry to `None`.

Suppose we remove key 226 from our hash table and set the entry at element 6 to `None`. What happens if we then perform a search for key 903? The `htSearch()` function will return `False`, indicating the key is not in the table, even though it's located at element 8. The reason for the unsuccessful search is due to element 6 containing a null reference from that key having been previously removed, as illustrated in Figure 11.9. Remember, key 903 maps to element 6 but when it was added, a new slot had to be found via a probe since key 226 already occupied that slot. If we simply remove key 226, there is no way to indicate we have to probe past this point when searching for other keys.

Instead of simply setting the corresponding table entry to `None`, we can use a special flag to indicate the entry is now empty but it had been previously occupied.

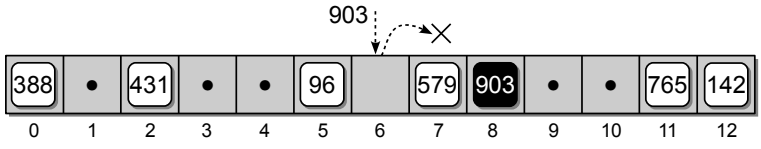

Figure 11.9: Incorrect deletion from the hash table.

Thus, when probing to add a new key or in searching for an existing key, we know the search must continue past the slot since the target may be stored beyond this point. Figure 11.10 illustrates the correct way to delete a key from the hash table. The delta Δ symbol is used to indicate a deleted entry.

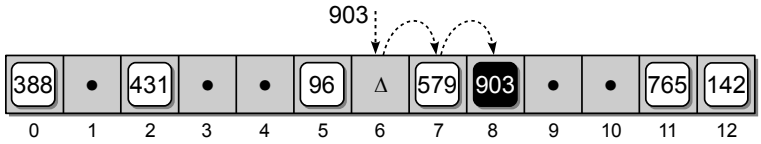

Figure 11.10: The correct way to delete a key from the hash table.

11.2.2 Clustering

As more keys are added to the hash table, more collisions are likely to occur. Since each collision requires a linear probe to find the next available slot, the keys begin to form ***clusters***. As the clusters grow larger, so too does the probability that the next key added to the table will result in a collision. If our table were empty, the probability of a key being added to any of the 13 empty slots is 1 out of 13, since it is equally likely the key can hash to any of the slots. Now consider the hash table in Figure 11.8. What is the probability the next key will occupy the empty slot at position 4? If the next key hashes to this position, it can be stored directly into the slot without the need to probe. This also results in a probability of 1 out of 13. But the probability the next key will occupy slot 9 is 5 out of 13. If the next key hashes to any of the slots between 5 and 9, it will be stored in slot 9 due to the linear probe required to find the first position beyond the cluster of keys. Thus, the key is five times more likely to occupy slot 9 than slot 4.

This type of clustering is known as ***primary clustering*** since it occurs near the original hash position. As the clusters grow larger, so too does the length of the search needed to find the next available slot. We can reduce the amount of primary clustering by changing the technique used in the probing. In this section, we examine several different probing techniques that can be employed to reduce primary clustering.

Modified Linear Probe

When probing to find the next available slot, a loop is used to iterate through the table entries. The order in which the entries are visited form a ***probe sequence***. The linear probe searches for the next available slot by stepping through the hash table entries in sequential order. The next array slot in the probe sequence can be represented as an equation:

$$\text{slot} = (\text{home} + i) \ \% \ M$$

where i is the ith probe in the sequence, $i = 1, 2, \ldots M - 1$. `home` is the ***home position***, which is the index to which the key was originally mapped by the hash function. The modulus operator is used to wrap back around to the front of the array after reaching the end. The use of the linear probe resulted in six collisions in our hash table of size $M = 13$:

```
h(765) => 11          h(579) => 7
h(431) => 2           h(226) => 5   => 6
h(96)  => 5           h(903) => 6   => 7  => 8 => 9
h(142) => 12          h(388) => 11  => 12 => 0
```

when the keys are inserted in the order:

```
765, 431, 96, 142, 579, 226, 903, 388
```

We can improve the linear probe by skipping over multiple elements instead of probing the immediate successor of each element. This can be done by changing the step size in the probe equation to some fixed constant c:

$$\text{slot} = (\text{home} + i * c) \ \% \ M$$

Suppose we use a linear probe with $c = 3$ to build the hash table using the same set of keys. This results in only two collisions as compared to six when $c = 1$ (the resulting hash table is illustrated in Figure 11.11):

```
h(765) => 11          h(579) => 7
h(431) => 2           h(226) => 5   => 8
h(96)  => 5           h(903) => 6
h(142) => 12          h(388) => 11  => 1
```

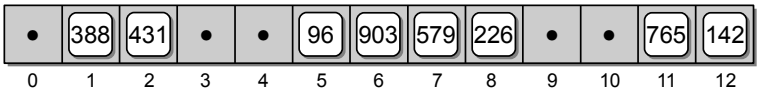

Figure 11.11: The hash table using a linear probe with $c = 3$.

Any value can be used for the constant factor, but to ensure the probe sequence includes all table entries, the constant factor c and the table size must be relatively prime. With a hash table of size $M = 13$, the linear probe with a constant factor $c = 2$ will visit every element. For example, if the key hashes to position 2, the table entries will be visited in the following order:

```
4, 6, 8, 10, 12, 1, 3, 5, 7, 9, 11, 0
```

If we use a value of $c = 3$, the probe sequence will be:

```
5, 8, 11, 1, 4, 7, 10, 0, 3, 6, 9, 12
```

Now, consider the case where the table size is $M = 10$ and the constant factor is $c = 2$. The probe sequence will only include the even numbered entries and will repeat the same sequence without possibly finding the key or an available entry to store a new key:

```
4, 6, 8, 0, 2, 4, 6, 8, 0
```

Quadratic Probing

The linear probe with a constant factor larger than 1 spreads the keys out from the initial hash position, but it can still result in clustering. The clusters simply move equal distance from the initial hash positions. A better approach for reducing primary clustering is with the use of ***quadratic probing***, which is specified by the equation:

$$\text{slot} = (\text{home} + i^2) \ \% \ M$$

Quadratic probing eliminates primary clustering by increasing the distance between each probe in the sequence. When used to build the hash table using the sample set of keys, we get seven collisions (the resulting hash table is illustrated in Figure 11.12):

```
h(765) => 11          h(579) => 7
h(431) => 2           h(226) => 5   => 6
h(96)  => 5           h(903) => 6   => 7  => 10
h(142) => 12          h(388) => 11  => 12 => 2  => 7 => 1
```

While the number of collisions has increased, the primary clustering has been reduced. In practice, quadratic probing typically reduces the number of collisions but introduces the problem of ***secondary clustering***. Secondary clustering occurs when two keys map to the same table entry and have the same probe sequence. For example, if we were to add key 648 to our table, it would hash to slot 11 and follow the same probe sequence as key 388. Finally, there is no guarantee the quadratic probe will visit every entry in the table. But if the table size is a prime number, at least half of the entries will be visited.

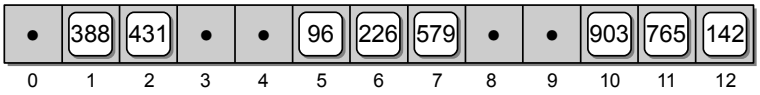

Figure 11.12: The hash table using a quadratic probe.

Double Hashing

The quadratic probe distributes the keys by increasing steps in the probe sequence. But the same sequence is followed by multiple keys that map to the same table

entry, which results in the secondary clustering. This occurs because the probe equation is based solely on the original hash slot. A better approach for reducing secondary clustering is to base the probe sequence on the key itself. In ***double hashing***, when a collision occurs, the key is hashed by a second function and the result is used as the constant factor in the linear probe:

$$\text{slot} = (\text{home} + i * hp(\text{key})) \ \% \ M$$

While the step size remains constant throughout the probe, multiple keys that map to the same table entry will have different probe sequences. To reduce clustering, the second hash function should not be the same as the main hash function and it should produce a valid index in the range $0 < c < M$. A simple choice for the second hash function takes the form:

$$hp(\text{key}) = 1 + \text{key} \ \% \ P$$

where P is some constant less than M. For example, suppose we define a second hash function:

$$hp(\text{key}) = 1 + \text{key} \ \% \ 8$$

and use it with double hashing to build a hash table from our sample keys. This results in only two collisions:

```
h(765) => 11          h(579) => 7
h(431) => 2           h(226) => 5   => 8
h(96)  => 5           h(903) => 6
h(142) => 12          h(388) => 11  => 3
```

The hash table resulting from the use of double hashing is illustrated in Figure 11.13. The double hashing technique is most commonly used to resolve collisions since it reduces both primary and secondary clustering. To ensure every table entry is visited during the probing, the table size must be a prime number. We leave it as an exercise to show why this is necessary.

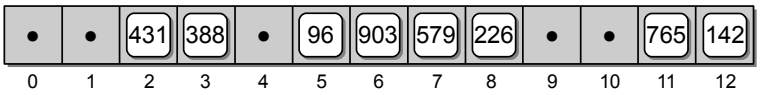

Figure 11.13: The hash table using double hashing.

11.2.3 Rehashing

We have looked at how to use and manage a hash table, but how do we decide how big the hash table should be? If we know the number of entries that will be

stored in the table, we can easily create a table large enough to hold the entire collection. In many instances, however, there is no way to know up front how many keys will be stored in the hash table. In this case, we can start with a table of some given size and then grow or expand the table as needed to make room for more entries. We used a similar approach with a vector. When all available slots in the underlying array had been consumed, a new larger array was created and the contents of the vector copied to the new array.

With a hash table, we create a new array larger than the original, but we cannot simply copy the contents from the old array to the new one. Instead, we have to rebuild or ***rehash*** the entire table by adding each key to the new array as if it were a new key being added for the first time. Remember, the search keys were added to the hash table based on the result of the hash function and the result of the function is based on the size of the table. If we increase the size of the table, the function will return different hash values and the keys may be stored in different entries than in the original table. For example, suppose we create a hash table of size $M = 17$ and insert our set of sample keys using a simple linear probe with $c = 1$. Applying the hash function to the keys yields the following results, which includes a single collision:

```
h(765) => 0                    h(579) => 1
h(431) => 6                    h(226) => 5
h(96)  => 11                   h(903) => 2
h(142) => 6  => 7              h(388) => 14
```

The original hash table using a linear probe is shown in Figure 11.14(a) and the new larger hash table is shown in Figure 11.14(b). You will notice the keys are stored in different locations due to the larger table size.

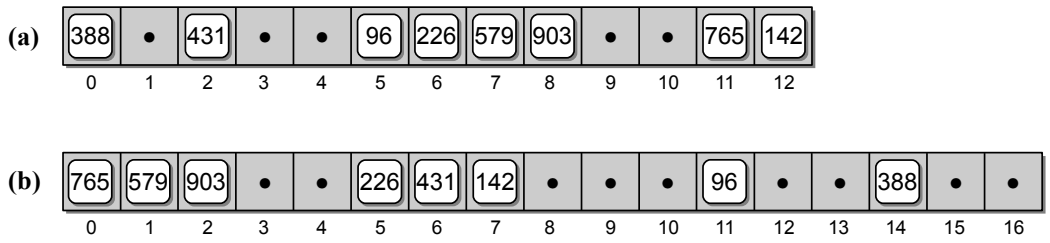

Figure 11.14: The result of enlarging the hash table from 13 elements to 17.

As the table becomes more full, the more likely it is that collisions will occur. Experience has shown that hashing works best when the table is no more than approximately three quarters full. Thus, if the hash table is to be expanded, it should be done before the table becomes full. The ratio between the number of keys in the hash table and the size of the table is called the ***load factor***. In practice, a hash table should be expanded before the load factor reaches 80%.

The amount by which the table should be expanded can depend on the application, but a good rule of thumb is to at least double its size. As we indicated earlier,

most of the probing techniques can benefit from a table size that is a prime number. To determine the actual size of the new table, we can first double the original size, $2m$ and then search for the first prime number greater than $2m$. Depending on the application and the type of probing used, you may be able to simply double the size and add one, $2m + 1$. Note that by adding one, the resulting size will be an odd number, which results in fewer divisors for the given table size.

11.2.4 Efficiency Analysis

The ultimate goal of hashing is to provide direct access to data items based on the search keys in the hash table. But, as we've seen, collisions routinely occur due to multiple keys mapping to the same table entry. The efficiency of the hash operations depends on the hash function, the size of the table, and the type of probe used to resolve collisions. The insertion and deletion operations both require a search to locate the slot into which a new key can be inserted or the slot containing the key to be deleted. Once the slot has been located, the insertion and deletion operations are simple and only require constant time. The time required to perform the search is the main contributor to the overall time of the three hash table operations: searching, insertions, and deletions.

To evaluate the search performed in hashing, assume there are n elements currently stored in the table of size m. In the best case, which only requires constant time, the key maps directly to the table entry containing the target and no collision occurs. When a collision occurs, however, a probe is required to find the target key. In the worst case, the probe has to visit every entry in the table, which requires $O(m)$ time.

From this analysis, it appears as if hashing is no better than a basic linear search, which also requires linear time. The difference, however, is that hashing is very efficient in the average case. The average case assumes the keys are uniformly distributed throughout the table. It depends on the average probe length and the average probe length depends on the load factor. Given the load factor $\alpha = \frac{n}{m} < 1$, Donald E. Knuth, author of the definitive book series on data structures and algorithms, *The Art of Computer Programming*, derived equations for the average probe length. The times depend on the type of probe used in the search and whether the search was successful.

When using a linear probe, the average number of comparisons required to locate a key in the hash table for a successful search is:

$$\frac{1}{2}\left(1 + \frac{1}{(1-\alpha)^2}\right)$$

and for an unsuccessful search:

$$\frac{1}{2}\left(1 + \frac{1}{(1-\alpha)}\right)$$

When using a quadratic probe or double hashing, the average number of comparisons required to locate a key for a successful search is:

$$\frac{-\log(1-\alpha)}{\alpha}$$

and for an unsuccessful search:

$$\frac{1}{(1-\alpha)}$$

Table 11.1 shows the average number of comparisons for both linear and quadratic probes when used with various load factors. As the load factor increases beyond approximately 2/3, the average number of comparisons become very large, especially for an unsuccessful search. The data in the table also shows that the quadratic and double hashing probes can allow for higher load factors than the linear probe.

Load Factor	**0.25**	**0.5**	**0.67**	**0.8**	**0.99**
Successful search:					
Linear Probe	1.17	1.50	2.02	3.00	50.50
Quadratic Probe	1.66	2.00	2.39	2.90	6.71
Unsuccessful search:					
Linear Probe	1.39	2.50	5.09	13.00	5000.50
Quadratic Probe	1.33	2.00	3.03	5.00	100.00

Table 11.1: Average search times for both linear and quadratic probes.

Based on experiments and the equations above, we can conclude that the hash operations only require an average time of $O(1)$ when the load factor is between 1/2 and 2/3. Compare this to the average times for the linear and binary searches ($O(n)$ and $O(\log n)$, respectively) and we find that hashing provides an efficient solution for the search operation.

11.3 Separate Chaining

When a collision occurs, we have to probe the hash table to find another available slot. In the previous section, we reviewed several probing techniques that can be used to help reduce the number of collisions. But we can eliminate collisions entirely if we allow multiple keys to share the same table entry. To accommodate multiple keys, linked lists can be used to store the individual keys that map to the same entry. The linked lists are commonly referred to as ***chains*** and this technique of collision resolution is known as ***separate chaining***.

In separate chaining, the hash table is constructed as an array of linked lists. The keys are mapped to an individual index in the usual way, but instead of storing

the key into the array elements, the keys are inserted into the linked list referenced from the corresponding entry; there's no need to probe for a different slot. New keys can be prepended to the linked list since the nodes are in no particular order. Figure 11.15 illustrates the use of separate chaining to build a hash table.

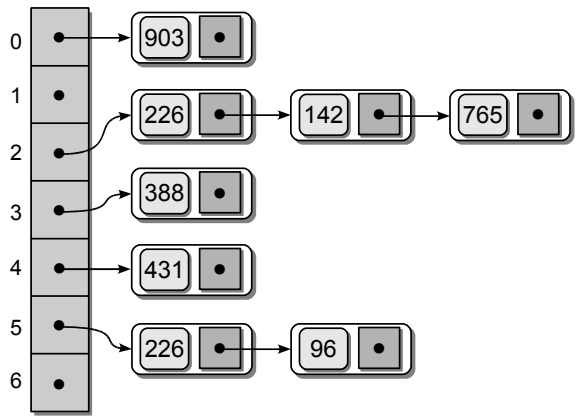

Figure 11.15: Hash table using separate chaining.

The search operation is much simpler when using separate chaining. After mapping the key to an entry in the table, the corresponding linked list is searched to determine if the key is in the table. When deleting a key, the key is again mapped in the usual way to find the linked list containing that key. After locating the list, the node containing the key is removed from the linked list just as if we were removing any other item from a linked list. Since the keys are not stored in the array elements themselves, we no longer have to mark the entry as having been filled by a previously deleted key.

Separate chaining is also known as ***open hashing*** since the keys are stored outside the table. The term ***closed hashing*** is used when the keys are stored within the elements of the table as described in the previous section. To confuse things a bit, some computer scientists also use the terms ***closed addressing*** to describe open hashing and ***open addressing*** to describe closed hashing. The use of the addressing terms refers to the possible locations of the keys in relation to the table entries. In open addressing, the keys may have been stored in an open slot different from the one to which it originally mapped while in closed addressing, the key is contained within the entry to which it mapped.

The table size used in separate chaining is not as important as in closed hashing since multiple keys can be stored in the various linked list. But it still requires attention since better key distribution can be achieved if the table size is a prime number. In addition, if the table is too small, the linked lists will grow larger with the addition of each new key. If the list become too large, the table can be rehashed just as we did when using closed hashing.

The analysis of the efficiency for separate chaining is similar to that of closed hashing. As before, the search required to locate a key is the most time consuming

part of the hash operations. Mapping a key to an entry in the hash table can be done in one step, but the time to search the corresponding linked list is based on the length of that list. In the worst case, the list will contain all of the keys stored in the hash table, resulting in a linear time search. As with closed hashing, separate chaining is very efficient in the average case. The average time to locate a key within the hash table assumes the keys are uniformly distributed across the table and it depends on the average length of the linked lists. If the hash table contains n keys and m entries, the average list length is $\frac{n}{m}$, which is the same as the load factor. Deriving equations for the average number of searches in separate chaining is much easier than with closed hashing. The average number of comparisons required to locate a key in the hash table for a successful search is:

$$1 + \frac{\alpha}{2}$$

and for an unsuccessful search is:

$$1 + \alpha$$

When the load factor is less than 2 (twice the number of keys as compared to the number of table entries), it can be shown that the hash operations only require $O(1)$ time in the average case. This is a better average time than that for closed hashing, which is an advantage of separate chaining. The drawback to separate chaining, however, is the need for additional storage used by the link fields in the nodes of the linked lists.

11.4 Hash Functions

The efficiency of hashing depends in large part on the selection of a good hash function. As we saw earlier, the purpose of a hash function is to map a set of search keys to a range of index values corresponding to entries in a hash table. A "perfect" hash function will map every key to a different table entry, resulting in no collisions. But this is seldom achieved except in cases like our collection of products in which the keys are within a small range or when the keys are known beforehand. Instead, we try to design a good hash function that will distribute the keys across the range of hash table indices as evenly as possible. There are several important guidelines to consider in designing or selecting a hash function:

- The computation should be simple in order to produce quick results.
- The resulting index cannot be random. When a hash function is applied multiple times to the same key, it must always return the same index value.
- If the key consists of multiple parts, every part should contribute in the computation of the resulting index value.
- The table size should be a prime number, especially when using the modulus operator. This can produce better distributions and fewer collisions as it tends to reduce the number of keys that share the same divisor.

Integer keys are the easiest to hash, but there are many times when we have to deal with keys that are either strings or a mixture of strings and integers. When dealing with non-integer keys, the most common approach is to first convert the key to an integer value and then apply an integer-based hash function to that value. In this section, we first explore several hash functions that can be used with integers and then look at common techniques used to convert strings to integer values that can then be hashed.

Division

The simplest hash function for integer values is the one we have been using throughout the chapter. The integer key, or a mixed type key that has been converted to an integer, is divided by the size of the hash table with the remainder becoming the hash table index:

$$h(\text{key}) = \text{key} \ \texttt{\%} \ M$$

Computing the remainder of an integer key is the easiest way to ensure the resulting index always falls within the legal range of indices. The division technique is one of the most commonly used hash functions, applied directly to an integer key or after converting a mixed type key to an integer.

Truncation

For large integers, some columns in the key value are ignored and not used in the computation of the hash table index. In this case, the index is formed by selecting the digits from specific columns and combining them into an integer within the legal range of indices. For example, if the keys are composed of integer values that all contain seven digits and the hash table size is 1000, we can concatenate the first, third, and sixth digits (counting from right to left) to form the index value. Using this technique, key value 4873152 would hash to index 812.

Folding

In this method, the key is split into multiple parts and then combined into a single integer value by adding or multiplying the individual parts. The resulting integer value is then either truncated or the division method is applied to fit it within the range of legal table entries. For example, given a key value 4873152 consisting of seven digits, we can split it into three smaller integer values (48, 731, and 52) and then sum these to obtain a new integer: `48 + 731 + 52 = 831`. The division method can then be used to obtain the hash table index. This method can also be used when the keys store data with explicit components such as social security numbers or phone numbers.

Hashing Strings

Strings can also be stored in a hash table. The string representation has to be converted to an integer value that can be used with the division or truncation

methods to generate an index within the valid range. There are many different techniques available for this conversion. The simplest approach is to sum the ASCII values of the individual characters. For example, if we use this method to hash the string `'hashing'`, the result will be:

```
104 + 97 + 115 + 104 + 105 + 110 + 103 = 738
```

This approach works well with small hash tables. But when used with larger tables, short strings will not hash to the larger index values; they will only be used when probed. For example, suppose we apply this method to strings containing seven characters, each with a maximum ASCII value of 127. Summing the ASCII values will yield a maximum value of `127 * 7 = 889`. A second approach that can provide good results regardless of the string length uses a polynomial:

$$s_0 a^{n-1} + s_1 a^{n-2} + \cdots + s_{n-3} a^2 + s_{n-2} a + s_{n-1}$$

where a is a non-zero constant, s_i is the ith element of the string, and n is the length of the string. If we use this method with the string `'hashing'`, where $a = 27$, the resulting hash value will be 41746817200. This value can then be used with the division method to yield an index value within the valid range.

11.5 The HashMap Abstract Data Type

One of the most common uses of a hash table is for the implementation of a map. In fact, Python's dictionary is implemented using a hash table with closed hashing. The definition of the Map ADT from Chapter 3 allows for the use of any type of comparable key, which differs from Python's dictionary since the latter requires the keys to be hashable. That requirement can limit the efficient use of the dictionary since we must define our own hash function for any user-defined types that are to be used as dictionary keys. Our hash function must produce good results or the dictionary operations may not be very efficient.

In this section, we provide an implementation for the map that is very similar to the approach used in implementing Python's dictionary. Since this version requires the keys to be hashable, we use the name HashMap to distinguish it from the more general Map ADT. For the implementation of the HashMap ADT, we are going to use a hash table with closed hashing and a double hashing probe. The source code is provided in Listing 11.1 on the next page.

The Hash Table

In implementing the HashMap ADT, we must first decide how big the hash table should be. The HashMap ADT is supposed to be a general purpose structure that can store any number of key/value pairs. To maintain this property, we must allow the hash table to expand as needed. Thus, we can start with a relatively small table ($M = 7$) and allow it to expand as needed by rehashing each time the load factor is exceeded. The next question we need to answer is what load factor

should we use? As we saw earlier, a load factor between 1/2 and 2/3 provides good performance in the average case. For our implementation we are going to use a load factor of 2/3.

Listing 11.1 The `hashmap.py` module.

```
# Implementation of the Map ADT using closed hashing and a probe with
# double hashing.
from arrays import Array

class HashMap :
   # Defines constants to represent the status of each table entry.
  UNUSED = None
  EMPTY = _MapEntry( None, None )

   # Creates an empty map instance.
  def __init__( self ):
    self._table = Array( 7 )
    self._count = 0
    self._maxCount = len(self._table) - len(self._table) // 3

   # Returns the number of entries in the map.
  def __len__( self ):
    return self._count

   # Determines if the map contains the given key.
  def __contains__( self, key ):
    slot = self._findSlot( key, False )
    return slot is not None

   # Adds a new entry to the map if the key does not exist. Otherwise, the
   # new value replaces the current value associated with the key.
  def add( self, key, value ):
    if key in self :
      slot = self._findSlot( key, False )
      self._table[slot].value = value
      return False
    else :
      slot = self._findSlot( key, True )
      self._table[slot] = _MapEntry( key, value )
      self._count += 1
      if self._count == self._maxCount :
        self._rehash()
      return True

   # Returns the value associated with the key.
  def valueOf( self, key ):
    slot = self._findSlot( key, False )
    assert slot is not None, "Invalid map key."
    return self._table[slot].value

   # Removes the entry associated with the key.
  def remove( self, key ):
    ......

```

```
50     # Returns an iterator for traversing the keys in the map.
51    def __iter__( self ):
52      ......
53
54     # Finds the slot containing the key or where the key can be added.
55     # forInsert indicates if the search is for an insertion, which locates
56     # the slot into which the new key can be added.
57    def _findSlot( self, key, forInsert ):
58       # Compute the home slot and the step size.
59      slot = self._hash1( key )
60      step = self._hash2( key )
61
62       # Probe for the key.
63      M = len(self._table)
64      while self._table[slot] is not UNUSED :
65        if forInsert and \
66           (self._table[slot] is UNUSED or self._table[slot] is EMPTY) :
67          return slot
68        elif not forInsert and \
69           (self._table[slot] is not EMPTY and self._table[slot].key == key) :
70          return slot
71        else :
72          slot = (slot + step) % M
73
74     # Rebuilds the hash table.
75    def _rehash( self ) :
76       # Create a new larger table.
77      origTable = self._table
78      newSize = len(self._table) * 2 + 1
79      self._table = Array( newSize )
80
81       # Modify the size attributes.
82      self._count = 0
83      self._maxCount = newSize - newSize // 3
84
85       # Add the keys from the original array to the new table.
86      for entry in origTable :
87        if entry is not UNUSED and entry is not EMPTY :
88          slot = self._findSlot( key, True )
89          self._table[slot] = entry
90          self._count += 1
91
92     # The main hash function for mapping keys to table entries.
93    def _hash1( self, key ):
94      return abs( hash(key) ) % len(self._table)
95
96     # The second hash function used with double hashing probes.
97    def _hash2( self, key ):
98      return 1 + abs( hash(key) ) % (len(self._table) - 2)
99
100 # Storage class for holding the key/value pairs.
101 class _MapEntry :
102   def __init__( self, key, value ):
103     self.key = key
104     self.value = value
```

In the constructor (lines 11–14), we create three attributes: `table` stores the array used for the hash table, `count` indicates the number of keys currently stored in the table, and `maxCount` indicates the maximum number of keys that can be stored in the table before exceeding the load factor. Instead of using floating-point operations to determine if the load factor has been exceeded, we can store the maximum number of keys needed to reach that point. Each time the table is expanded, a new value of `maxCount` is computed. For the initial table size of 7, this value will be 5.

The key/value entries can be stored in the table using the same storage class `_MapEntry` as used in our earlier implementation. But we also need a way to flag an entry as having been previously used by a key but has now been deleted. The easiest way to do this is with the use of a dummy `_MapEntry` object. When a key is deleted, we simply store an alias of the dummy object reference in the corresponding table entry. For easier readability of the source code, we create two named constants in lines 7–8 to indicate the two special states for the table entries: an `UNUSED` entry, which is indicated by a null reference, is one that has not yet been used to store a key; and an `EMPTY` entry is one that had previously stored a key but has now been deleted. The third possible state of an entry, which is easily determined if the entry is not one of the other two states, is one that is currently occupied by a key.

Hash Functions

Our implementation will need two hash functions: the main function for mapping the key to a home position and the function used with the double hashing. For both functions, we are going to use the simple division method in which the key value is divided by the size of the table and the remainder becomes the index to which the key maps. The division hash functions defined earlier in the chapter assumed the search key is an integer value. But the HashMap ADT allows for the storage of any type of search key, which includes strings, floating-point values, and even user-defined types. To accommodate keys of various data types, we can use Python's built-in `hash()` function, which is automatically defined for all of the built-in types. It hashes the given key and returns an integer value that can be used in the division method. But the value returned by the Python's `hash()` function can be any integer, not just positive values or those within a given range. We can still use the function and simply take its absolute value and then divide it by the size of the table. The main hash function for our implementation is defined as:

$$h(\text{key}) = |\texttt{hash}(\text{key})| \ \texttt{\%} \ M$$

while the second function for use with double hashing is defined as:

$$hp(\text{key}) = 1 + |\texttt{hash}(\text{key})| \ \texttt{\%} \ (M - 2)$$

The size of our hash table will always be an odd number, so we subtract 2 from the size of the table in the second function to ensure the division is by an odd number. The two hash functions are implemented in lines 93–98 of Listing 11.1.

To use objects of a user-defined class as keys in the dictionary, the class must implement both the `__hash__` and `__eq__` methods. The `__hash__` method should hash the contents of the object and return an integer that can be used by either of our two hash functions, $h()$ and $hp()$. The `__eq__` is needed for the equality comparison in line 69 of Listing 11.1, which determines if the key stored in the given slot is the target key.

Searching

As we have seen, a search has to be performed no matter which hash table operation we use. To aide in the search, we create the `_findSlot()` helper method as shown in lines 57–72. Searching the table to determine if a key is simply contained in the table and searching for a key to be deleted require the same sequence of steps. After mapping the key to its home position, we determine if the key was found at this location or if a probe has to be performed. When probing, we step through the keys using the step size returned by the second hash function. The probe continues until the key has been located or we encounter an unused slot (contains a null reference). The search used to locate a slot for the insertion of a new key, however, has one major difference. The probe must also terminate if we encounter a table entry marked as empty from a previously deleted key since a new key can be stored in such an entry.

This minor difference between the two types of searches is handled by the `forInsert` argument. When `True`, a search is performed for the location where a new key can be inserted and the index of that location is returned. When the argument is `False`, a normal search is performed and either the index of the entry containing the key is returned or `None` is returned when the key is not in the table. When used in the `__contains__` and `valueOf()` methods, the `_findSlot()` method is called with a value of `False` for the `forInsert` argument.

Insertions

The `add()` method also uses the `_findSlot()` helper method. In fact, it's called twice. First, we determine if the key is in the table that indirectly calls the `__contains__` method. If the key is in the table, we have to locate the key through a normal search and modify its corresponding value. On the other, if the key is not in the table, `__findSlot__` is called with a value of `True` passed to the `forInsert` argument to locate the next available slot. Finally, if the key is new and has to be added to the table, we check the count and determine if it exceeds the load factor, in which case the table has to be rehashed. The remove operation and the implementation of an iterator for use with this new version of the Map ADT are left as exercises.

Rehashing

The rehash operation is shown in lines 75–90 of Listing 11.1. The first step is to create a new larger array. For simplicity, the new size is computed to be $M * 2 + 1$,

which ensures an odd value. A more efficient solution would ensure the new size is always a prime number by searching for the next prime number larger than $M * 2 + 1$.

The original array is saved in a temporary variable and the new array is assigned to the `table` attribute. The reason for assigning the new array to the attribute at this time is that we will need to use the `_findSlot()` method to add the keys to the new array and that method works off the `table` attribute. The `count` and `maxCount` are also reset. The value of `maxCount` is set to be approximately two-thirds the size of the new table using the expression shown in line 83 of Listing 11.1.

Finally, the key/value pairs are added to the new array, one at a time. Instead of using the `add()` method, which first verifies the key is new, we perform the insertion of each directly within the `for` loop.

11.6 Application: Histograms

Graphical displays or charts of tabulated frequencies are very common in statistics. These charts, known as ***histograms***, are used to show the distribution of data across discrete categories. A histogram consists of a collection of categories and counters. The number and types of categories can vary depending on the problem. The counters are used to accumulate the number of occurrences of values within each category for a given data collection. Consider the example histogram in Figure 11.16. The five letter grades are the categories and the heights of the bars represent the value of the counters.

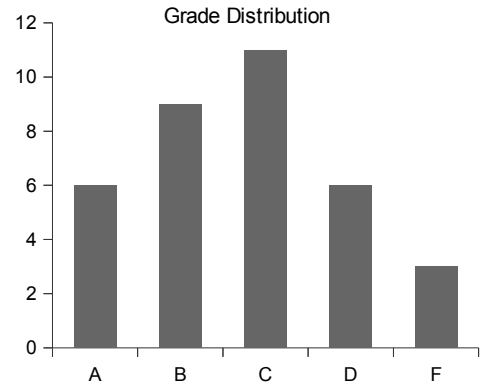

Figure 11.16: Sample histogram for a distribution of grades.

11.6.1 The Histogram Abstract Data Type

We can define an abstract data type for collecting and storing the frequency counts used in constructing a histogram. An ideal ADT would allow for building a general purpose histogram that can contain many different categories and be used with many different problems.

Define **Histogram ADT**

A *histogram* is a container that can be used to collect and store discrete frequency counts across multiple categories representing a distribution of data. The category objects must be comparable.

- `Histogram( catSeq )`: Creates a new histogram containing the categories provided in the given sequence, `catSeq`. The frequency counts of the categories are initialized to zero.
- `getCount( category )`: Returns the frequency count for the given category, which must be valid.
- `incCount( category )`: Increments the count by 1 for the given category. The supplied category must be valid.
- `totalCount()`: Returns a sum of the frequency counts for all of the categories.
- *`iterator`*`()`: Creates and returns an iterator for traversing over the histogram categories.

Building a Histogram

The program in Listing 11.2 produces a text-based version of the histogram from Figure 11.16 and illustrates the use of the Histogram ADT. The program extracts a collection of numeric grades from a text file and assigns a letter grade to each value based on the common 10-point scale: A: 100 – 90, B: 89 – 80, C: 79 – 70, D: 69 – 60, F: 59 – 0. The frequency counts of the letter grades are tabulated and then used to produce a histogram.

Listing 11.2 The `buildhist.py` program.

```
# Prints a histogram for a distribution of letter grades computed
# from a collection of numeric grades extracted from a text file.

from maphist import Histogram

def main():
   # Create a Histogram instance for computing the frequencies.
  gradeHist = Histogram( "ABCDF" )

   # Open the text file containing the grades.
  gradeFile = open('cs101grades.txt', "r")

   # Extract the grades and increment the appropriate counter.
  for line in gradeFile :
    grade = int(line)
    gradeHist.incCount( letterGrade(grade) )

```

(Listing Continued)

Listing 11.2 Continued ...

```
18     # Print the histogram chart.
19    printChart( gradeHist )
20
21  # Determines the letter grade for the given numeric value.
22  def letterGrade( grade ):
23    if grade >= 90 :
24      return 'A'
25    elif grade >= 80 :
26      return 'B'
27    elif grade >= 70 :
28      return 'C'
29    elif grade >= 60 :
30      return 'D'
31    else :
32      return 'F'
33
34  # Prints the histogram as a horizontal bar chart.
35  def printChart( gradeHist ):
36    print( "           Grade Distribution" )
37     # Print the body of the chart.
38    letterGrades = ( 'A', 'B', 'C', 'D', 'F' )
39    for letter in letterGrades :
40      print( "  |" )
41      print( letter + " +", end = "" )
42      freq = gradeHist.getCount( letter )
43      print( '*' * freq )
44
45     # Print the x-axis.
46    print( "  |" )
47    print( "  +----+----+----+----+----+----+----+----" )
48    print( "  0    5    10   15   20   25   30   35" )
49
50  # Calls the main routine.
51  main()
```

The `buildhist.py` program consists of three functions. The `main()` function drives the program, which extracts the numeric grades and builds an instance of the Histogram ADT. It initializes the histogram to contain the five letter grades as its categories. The `letterGrade()` function is a helper function, which simply returns the letter grade for the given numeric value. The `printChart()` function prints the text-based histogram using the frequency counts computed in the main routine. Assuming the following grades are extracted from the text file:

```
77 89 53 95 68 86 91 89 60 70 80 77 73 73 93 85 83 67 75 71 94 64
79 97 59 69 61 80 73 70 82 86 70 45 100
```

the `buildhist.py` program would produce the following text-based histogram:

```
          Grade Distribution
  |
A +******
  |
B +*********
  |
C +**********
  |
D +******
  |
F +***
  |
  +----+----+----+----+----+----+----+----
  0    5    10   15   20   25   30   35
```

Implementation

To implement the Histogram ADT, we must select an appropriate data structure for storing the categories and corresponding frequency counts. There are several different structures and approaches that can be used, but the Map ADT provides an ideal solution since it already stores key/value mappings and allows for a full implementation of the Histogram ADT. To use a map, the categories can be stored in the key part of the key/value pairs and a counter (integer value) can be stored in the value part. When a category counter is incremented, the entry is located by its key and the corresponding value can be incremented and stored back into the entry. The implementation of the Histogram ADT using an instance of the hash table version of the Map ADT as the underlying structure is provided in Listing 11.3.

Listing 11.3 The `maphist.py` module.

```
1  # Implementation of the Histogram ADT using a Hash Map.
2
3  from hashmap import HashMap
4
5  class Histogram :
6     # Creates a histogram containing the given categories.
7    def __init__( self, catSeq ):
8      self._freqCounts = HashMap()
9      for cat in catSeq :
10       self._freqCounts.add( cat, 0 )
11
12    # Returns the frequency count for the given category.
13   def getCount( self, category ):
14     assert category in self._freqCounts, "Invalid histogram category."
15     return self._freqCounts.valueOf( category )
16
```

(Listing Continued)

Listing 11.3 Continued ...

```
17    # Increments the counter for the given category.
18    def incCount( self, category ):
19      assert category in self._freqCounts, "Invalid histogram category."
20      value = self._freqCounts.valueOf( category )
21      self._freqCounts.add( category, value + 1 )
22
23    # Returns the sum of the frequency counts.
24    def totalCount( self ):
25      total = 0
26      for cat in self._freqCounts :
27        total += self._freqCounts.valueOf( cat )
28      return total
29
30    # Returns an iterator for traversing the categories.
31    def __iter__( self ):
32      return iter( self._freqCounts )
```

The iterator operation defined by the ADT is implemented in lines 31–32. In Section 1.4.1, we indicated the iterator method is supposed to create and return an iterator object that can be used with the given collection. Since the Map ADT already provides an iterator for traversing over the keys, we can have Python access and return that iterator as if we had created our own. This is done using the `iter()` function, as shown in our implementation of the `__iter__` method in lines 31–32.

11.6.2 The Color Histogram

A histogram is used to tabulate the frequencies of multiple discrete categories. The Histogram ADT from the previous section works well when the collection of categories is small. Some applications, however, may deal with millions of distinct categories, none of which are known up front, and require a specialized version of the histogram. One such example is the ***color histogram***, which is used to tabulate the frequency counts of individual colors within a digital image. Color histograms are used in areas of image processing and digital photography for image classification, object identification, and image manipulation.

Color histograms can be constructed for any color space, but we limit our discussion to the more common discrete RGB color space. In the RGB color space, individual colors are specified by intensity values for the three primary colors: red, green, and blue. This color space is commonly used in computer applications and computer graphics because it is very convenient for modeling the human visual system. The intensity values in the RGB color space, also referred to as color components, can be specified using either real values in the range $[0 \ldots 1]$ or discrete values in the range $[0 \ldots 255]$. The discrete version is the most commonly used for the storage of digital images, especially those produced by digital cameras and scanners. With discrete values for the three color components, more than 16.7 million colors can be represented, far more than humans are capable of distinguishing. A value of 0 indicates no intensity for the given component while

255 indicates full intensity. Thus, white is represented with all three components set to 255, while black is represented with all three components set to 0.

We can define an abstract data type for a color histogram that closely follows that of the general histogram:

Define **Color Histogram ADT**

A *color histogram* is a container that can be used to collect and store frequency counts for multiple discrete RGB colors.

- `ColorHistogram()`: Creates a new empty color histogram.
- `getCount(red, green, blue)`: Returns the frequency count for the given RGB color, which must be valid.
- `incCount(red, green, blue)`: Increments the count by 1 for the given RGB color if the color was previously added to the histogram or the color is added to the histogram as a new entry with a count of 1.
- `totalCount()`: Returns a sum of the frequency counts for all colors in the histogram.
- *`iterator`*`()`: Creates and returns an iterator for traversing over the colors in the color histogram.

There are a number of ways we can construct a color histogram, but we need a fast and memory-efficient approach. The easiest approach would be to use a three-dimensional array of size $256 \times 256 \times 256$, where each element of the array represents a single color. This approach, however, is far too costly. It would require 256^3 array elements, most of which would go unused. On the other hand, the advantage of using an array is that accessing and updating a particular color is direct and requires no costly operations.

Other options include the use of a Python list or a linked list. But these would be inefficient when working with images containing millions of colors. In this chapter, we've seen that hashing can be a very efficient technique when used with a good hash function. For the color histogram, closed hashing would not be an ideal choice since it may require multiple rehashes involving hundreds of thousands, if not millions, of colors. Separate chaining can be used with good results, but it requires the design of a good hash function and the selection of an appropriately sized hash table.

A different approach can be used that combines the advantages of the direct access of the 3-D array and the limited memory use and fast searches possible with hashing and separate chaining. Instead of using a 1-D array to store the separate chains, we can use a 2-D array of size 256×256. The colors can be mapped to a specific chain by having the rows correspond to the red color component and the columns correspond to the green color component. Thus, all colors having the

same red and green components will be stored in the same chain, with only the blue components differing. Figure 11.17 illustrates this 2-D array of linked lists.

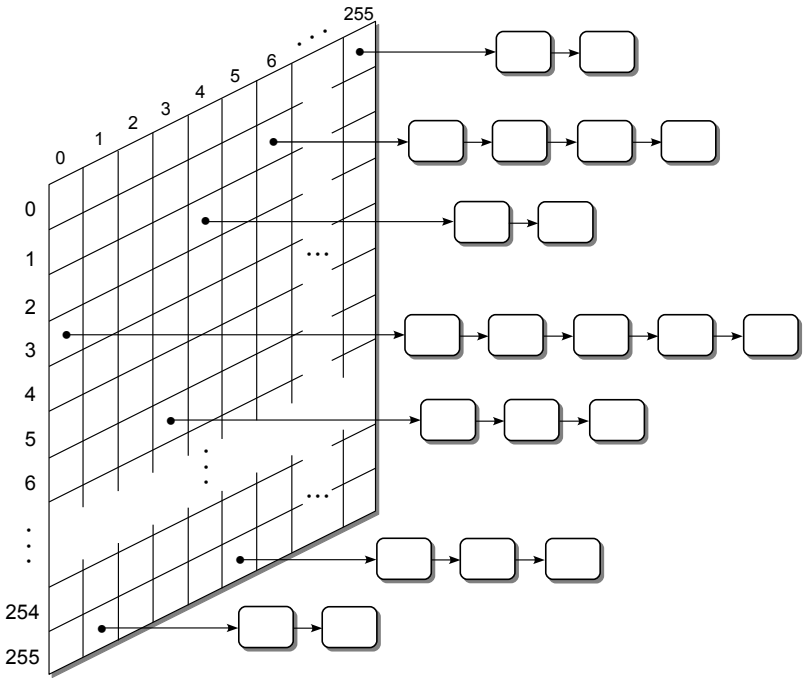

Figure 11.17: A 2-D array of linked lists used to store color counts in a color histogram.

Given a digital image consisting of n distinct pixels, all of which may contain unique colors, the histogram can be constructed in linear time. This time is derived from the fact that searching for the existence of a color can be done in constant time. Locating the specific 2-D array entry in which the color should be stored is a direct mapping to the corresponding array indices. Determining if the given color is contained in the corresponding linked list requires a linear search over the entire list. Since all of the nodes in the linked list store colors containing the same red and green components, they only differ in their blue components. Given that there are only 256 different blue component values, the list can never contain more than 256 entries. Thus, the length of the linked list is independent of the number of pixels in the image. This results in a worst case time of $O(1)$ to search for the existence of a color in the histogram in order to increment its count or to add a new color to the histogram. A search is required for each of the n distinct image pixels, resulting in a total time $O(n)$ in the worst case.

After the histogram is constructed, a traversal over the unique colors contained in the histogram is commonly performed. We could traverse over the entire 2-D array, one element at a time, and then traverse the linked list referenced from the individual elements. But this can be time consuming since in practice, many of the elements will not contain any colors. Instead, we can maintain a single separate linked list that contains the individual nodes from the various hash chains, as illustrated in Figure 11.18. When a new color is added to the histogram, a node is

created and stored in the corresponding chain. If we were to include a second link within the same nodes used in the chains to store the colors and color counts, we can then easily add each node to a separate linked list. This list can then be used to provide a complete traversal over the entries in the histogram without wasting time in visiting the empty elements of the 2-D array. The implementation of the color histogram is left as an exercise.

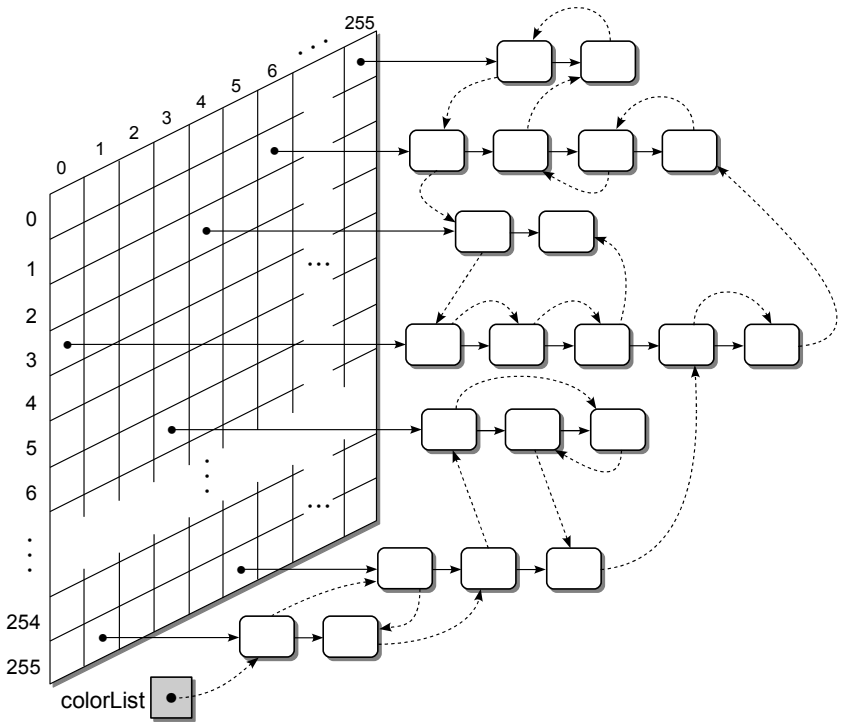

Figure 11.18: The individual chain nodes are linked together for faster traversals.

Exercises

11.1 Assume an initially empty hash table with 11 entries in which the hash function uses the division method. Show the contents of the hash table after the following keys are inserted (in the order listed), assuming the indicated type of probe is used: $67, 815, 45, 39, 2, 901, 34$.

(a) linear probe (with $c = 1$)

(b) linear probe (with $c = 3$)

(c) quadratic probe

(d) double hashing [with $hp(\text{key}) = (\text{key} * 3)$ `%` 7]

(e) separate chaining

11.2 Do the same as in Exercise 11.1 but use the following hash function to map the keys to the table entries:

$$h(\text{key}) = (2 * \text{key} + 3) \ \% \ 11$$

11.3 Show the contents of the hash table from Exercise 11.1 after rehashing with a new table containing 19 entries.

11.4 Consider a hash table of size 501 that contains 85 keys.

(a) What is the load factor?

(b) What is the average number of comparisons required to determine if the collection contains the key 73, if:

i. linear probing is used

ii. quadratic probing is used

iii. separate chaining is used

11.5 Do the same as in Exercise 11.4 but for a hash table of size 2031 that contains 999 keys.

11.6 Show why the table size must be a prime number in order for double hashing to visit every entry during the probe.

11.7 Design a hash function that can be used to map the two-character state abbreviations (including the one for the District of Columbia) to entries in a hash table that results in no more than three collisions when used with a table where $M < 100$.

Programming Projects

11.1 Implement the remove operation for the HashMap ADT.

11.2 Design and implement an iterator for use with the implementation of the HashMap ADT.

11.3 Modify the implementation of the HashMap ADT to:

(a) Use linear probing instead of double hashing

(b) Use quadratic probing instead of double hashing

(c) Use separate chaining instead of closed hashing

11.4 Design and implement a program that compares the use of linear probing, quadratic probing, and double hashing on a collection of string keys of varying lengths. The program should extract a collection of strings from a text file and compute the average number of collisions and the average number of probes.

11.5 Implement the Color Histogram ADT using the 2-D array of chains as described in the chapter.