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.

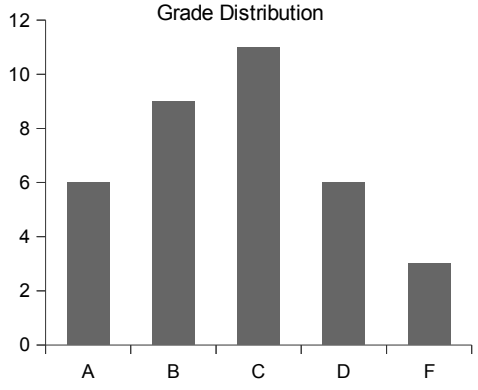

Figure 11.16: Sample histogram for a distribution of grades.

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.

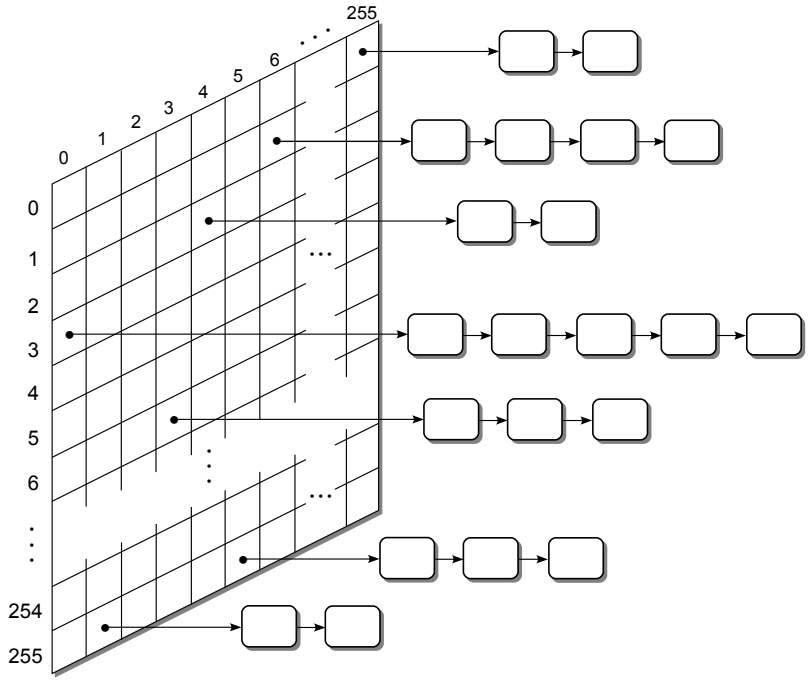

Figure 11.17: A 2-D array of linked lists used to store color counts in a color histogram.

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.

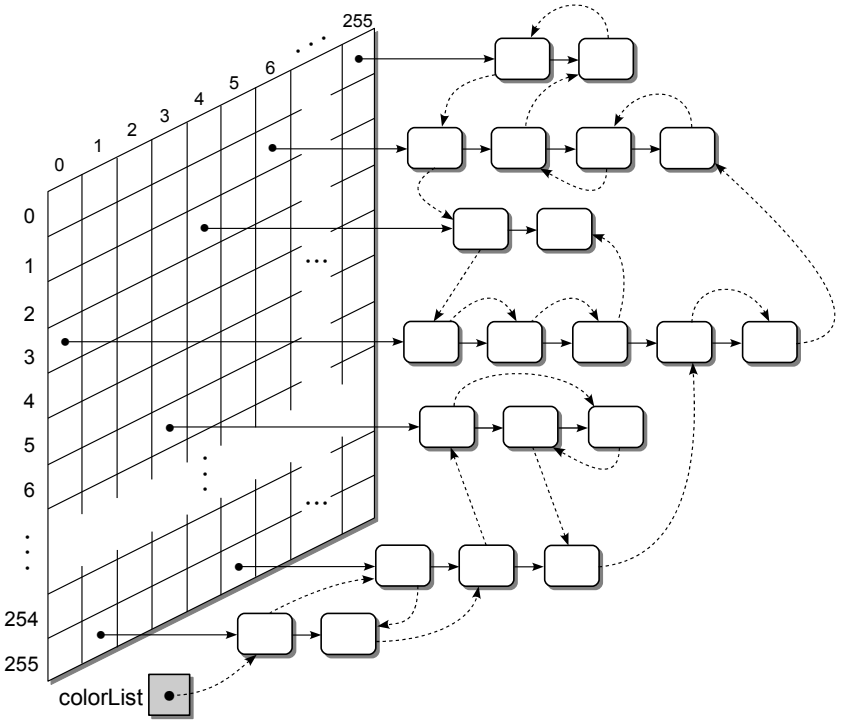

Figure 11.18: The individual chain nodes are linked together for faster traversals.

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