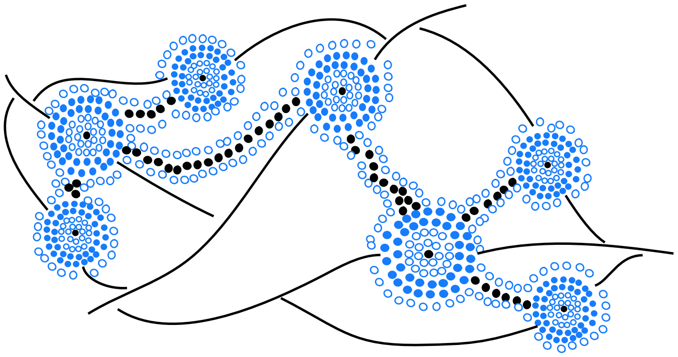
Chapter

3 Fundamental Data Structures



Contents

3.1 Using Arrays

In this section, we explore a few applications of arrays—the concrete data structures introduced in Section 1.3 that access their entries using integer indices.

3.1.1 Storing Game Entries in an Array

The first application we study is storing a sequence of high score entries for a video game in an array. This is representative of many applications in which a sequence of objects must be stored. We could just as easily have chosen to store records for patients in a hospital or the names of players on a football team. Nevertheless, let us focus on storing high score entries, which is a simple application that is already rich enough to present some important data-structuring concepts.

To begin, we consider what information to include in an object representing a high score entry. Obviously, one component to include is an integer representing the score itself, which we identify as score. Another useful thing to include is the name of the person earning this score, which we identify as name. We could go on from here, adding fields representing the date the score was earned or game statistics that led to that score. However, we omit such details to keep our example simple. A Java class, GameEntry, representing a game entry, is given in Code Fragment 3.1.

```java
public class GameEntry {
  private String name;                     // name of the person earning this score
  private int score;                       // the score value
  /** Constructs a game entry with given parameters.. */
  public GameEntry(String n, int s) {
    name = n;
    score = s;
  }
  /** Returns the name field. */
  public String getName( ) { return name; }
  /** Returns the score field. */
  public int getScore( ) { return score; }
  /** Returns a string representation of this entry. */
  public String toString( ) {
    return "(" + name + ", " + score + ")";
  }
}
```

Code Fragment 3.1: Java code for a simple GameEntry class. Note that we include methods for returning the name and score for a game entry object, as well as a method for returning a string representation of this entry.

A Class for High Scores

To maintain a sequence of high scores, we develop a class named Scoreboard. A scoreboard is limited to a certain number of high scores that can be saved; once that limit is reached, a new score only qualifies for the scoreboard if it is strictly higher than the lowest "high score" on the board. The length of the desired scoreboard may depend on the game, perhaps 10, 50, or 500. Since that limit may vary, we allow it to be specified as a parameter to our Scoreboard constructor.

Internally, we will use an array named board to manage the GameEntry instances that represent the high scores. The array is allocated with the specified maximum capacity, but all entries are initially **null**. As entries are added, we will maintain them from highest to lowest score, starting at index 0 of the array. We illustrate a typical state of the data structure in Figure 3.1, and give Java code to construct such a data structure in Code Fragment 3.2.

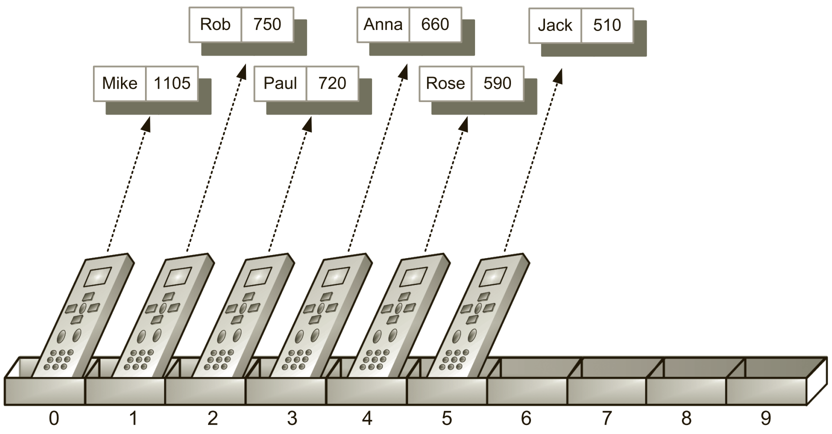


Figure 3.1: An illustration of an array of length ten storing references to six GameEntry objects in the cells with indices 0 to 5; the rest are **null** references.

```
/** Class for storing high scores in an array in nondecreasing order. */
public class Scoreboard {
  private int numEntries = 0;              // number of actual entries
  private GameEntry[ ] board;              // array of game entries (names & scores)
  /** Constructs an empty scoreboard with the given capacity for storing entries. */
  public Scoreboard(int capacity) {
    board = new GameEntry[capacity];
  }
  // more methods will go here
}
```

Code Fragment 3.2: The beginning of a Scoreboard class for maintaining a set of scores as GameEntry objects. (Completed in Code Fragments 3.3 and 3.4.)

Adding an Entry

One of the most common updates we might want to make to a Scoreboard is to add a new entry. Keep in mind that not every entry will necessarily qualify as a high score. If the board is not yet full, any new entry will be retained. Once the board is full, a new entry is only retained if it is strictly better than one of the other scores, in particular, the last entry of the scoreboard, which is the lowest of the high scores.

Code Fragment 3.3 provides an implementation of an update method for the Scoreboard class that considers the addition of a new game entry.

```
 9  /** Attempt to add a new score to the collection (if it is high enough) */
10  public void add(GameEntry e) {
11    int newScore = e.getScore( );
12    // is the new entry e really a high score?
13    if (numEntries < board.length || newScore > board[numEntries−1].getScore( )) {
14      if (numEntries < board.length)                // no score drops from the board
15        numEntries++;                               // so overall number increases
16      // shift any lower scores rightward to make room for the new entry
17      int j = numEntries − 1;
18      while (j > 0 && board[j−1].getScore( ) < newScore) {
19        board[j] = board[j−1];                      // shift entry from j-1 to j
20        j−−;                                        // and decrement j
21      }
22      board[j] = e;                                 // when done, add new entry
23    }
24  }
```

Code Fragment 3.3: Java code for inserting a GameEntry object into a Scoreboard.

When a new score is considered, the first goal is to determine whether it qualifies as a high score. This will be the case (see line 13) if the scoreboard is below its capacity, or if the new score is strictly higher than the lowest score on the board.

Once it has been determined that a new entry should be kept, there are two remaining tasks: (1) properly update the number of entries, and (2) place the new entry in the appropriate location, shifting entries with inferior scores as needed.

The first of these tasks is easily handled at lines 14 and 15, as the total number of entries can only be increased if the board is not yet at full capacity. (When full, the addition of a new entry will be counteracted by the removal of the entry with lowest score.)

The placement of the new entry is implemented by lines 17–22. Index j is initially set to numEntries − 1, which is the index at which the last GameEntry will reside after completing the operation. Either j is the correct index for the newest entry, or one or more immediately before it will have lesser scores. The while loop checks the compound condition, shifting entries rightward and decrementing j, as long as there is another entry at index $j-1$ with a score less than the new score.

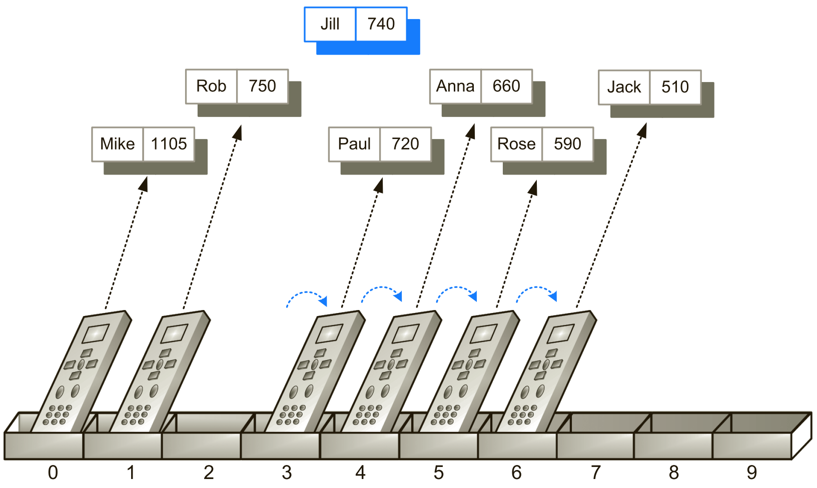


Figure 3.2: Preparing to add Jill's GameEntry object to the board array. In order to make room for the new reference, we have to shift any references to game entries with smaller scores than the new one to the right by one cell.

Figure 3.2 shows an example of the process, just after the shifting of existing entries, but before adding the new entry. When the loop completes, j will be the correct index for the new entry. Figure 3.3 shows the result of a complete operation, after the assignment of board[j] = e, accomplished by line 22 of the code.

In Exercise C-3.19, we explore how game entry addition might be simplified for the case when we don't need to preserve relative orders.

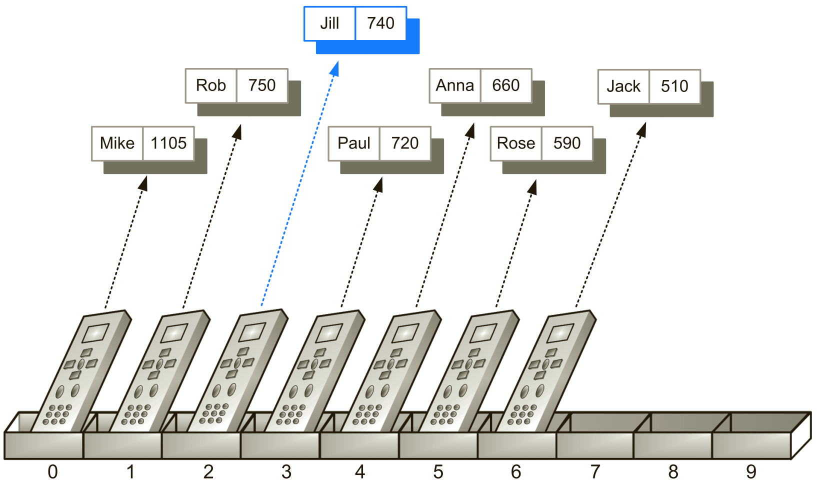


Figure 3.3: Adding a reference to Jill's GameEntry object to the board array. The reference can now be inserted at index 2, since we have shifted all references to GameEntry objects with scores less than the new one to the right.

Removing an Entry

Suppose some hot shot plays our video game and gets his or her name on our high score list, but we later learn that cheating occurred. In this case, we might want to have a method that lets us remove a game entry from the list of high scores. Therefore, let us consider how we might remove a reference to a GameEntry object from a Scoreboard.

We choose to add a method to the Scoreboard class, with signature remove(i), where i designates the current index of the entry that should be removed and returned. When a score is removed, any lower scores will be shifted upward, to fill in for the removed entry. If index i is outside the range of current entries, the method will throw an IndexOutOfBoundsException.

Our implementation for remove will involve a loop for shifting entries, much like our algorithm for addition, but in reverse. To remove the reference to the object at index i, we start at index i and move all the references at indices higher than i one cell to the left. (See Figure 3.4.)

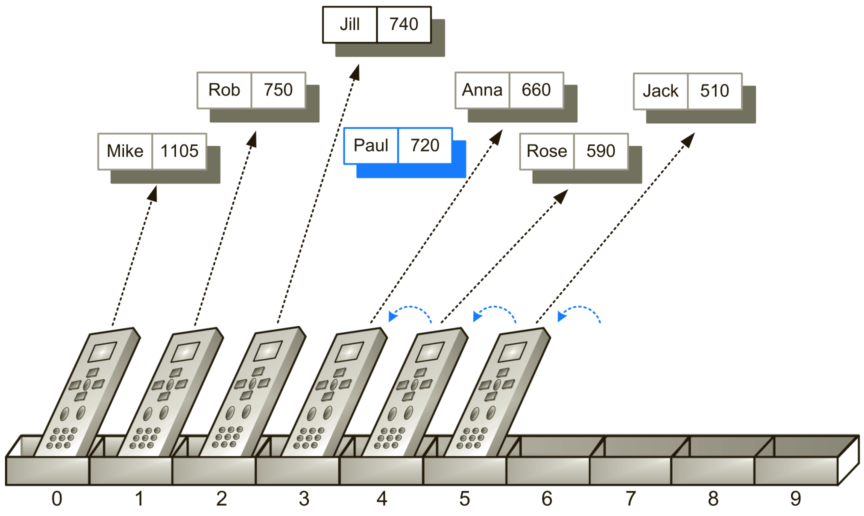


Figure 3.4: An illustration of the removal of Paul's score from index 3 of an array storing references to GameEntry objects.

Our implementation of the remove method for the Scoreboard class is given in Code Fragment 3.4. The details for doing the remove operation contain a few subtle points. The first is that, in order to remove and return the game entry (let's call it e) at index i in our array, we must first save e in a temporary variable. We will use this variable to return e when we are done removing it.

```
/** Remove and return the high score at index i. */
public GameEntry remove(int i) throws IndexOutOfBoundsException {
  if (i < 0 || i >= numEntries)
    throw new IndexOutOfBoundsException("Invalid index: " + i);
  GameEntry temp = board[i];                      // save the object to be removed
  for (int j = i; j < numEntries − 1; j++)        // count up from i (not down)
    board[j] = board[j+1];                        // move one cell to the left
  board[numEntries −1 ] = null;                   // null out the old last score
  numEntries−−;
  return temp;                                    // return the removed object
}
```

Code Fragment 3.4: Java code for performing the Scoreboard.remove operation.

The second subtle point is that, in moving references higher than *i* one cell to the left, we don't go all the way to the end of the array. First, we base our loop on the number of current entries, not the capacity of the array, because there is no reason for "shifting" a series of **null** references that may be at the end of the array. We also carefully define the loop condition, j < numEntries − 1, so that the last iteration of the loop assigns board[numEntries−2] = board[numEntries−1]. There is no entry to shift into cell board[numEntries−1], so we return that cell to **null** just after the loop. We conclude by returning a reference to the removed entry (which no longer has any reference pointing to it within the board array).

Conclusions

In the version of the Scoreboard class that is available online, we include an implementation of the toString( ) method, which allows us to display the contents of the current scoreboard, separated by commas. We also include a main method that performs a basic test of the class.

The methods for adding and removing objects in an array of high scores are simple. Nevertheless, they form the basis of techniques that are used repeatedly to build more sophisticated data structures. These other structures may be more general than the array structure above, of course, and often they will have a lot more operations that they can perform than just add and remove. But studying the concrete array data structure, as we are doing now, is a great starting point to understanding these other structures, since every data structure has to be implemented using concrete means.

In fact, later in this book, we will study a Java collections class, ArrayList, which is more general than the array structure we are studying here. The ArrayList has methods to operate on an underlying array; yet it also eliminates the error that occurs when adding an object to a full array by automatically copying the objects into a larger array when necessary. We will discuss the ArrayList class in far more detail in Section 7.2.

3.1.2 Sorting an Array

In the previous subsection, we considered an application for which we added an object to an array at a given position while shifting other elements so as to keep the previous order intact. In this section, we use a similar technique to solve the ***sorting*** problem, that is, starting with an unordered array of elements and rearranging them into nondecreasing order.

The Insertion-Sort Algorithm

We study several sorting algorithms in this book, most of which are described in Chapter 12. As a warm-up, in this section we describe a simple sorting algorithm known as ***insertion-sort***. The algorithm proceeds by considering one element at a time, placing the element in the correct order relative to those before it. We start with the first element in the array, which is trivially sorted by itself. When considering the next element in the array, if it is smaller than the first, we swap them. Next we consider the third element in the array, swapping it leftward until it is in its proper order relative to the first two elements. We continue in this manner with the fourth element, the fifth, and so on, until the whole array is sorted. We can express the insertion-sort algorithm in pseudocode, as shown in Code Fragment 3.5.

```
Algorithm InsertionSort($A$):
    Input: An array $A$ of $n$ comparable elements
    Output: The array $A$ with elements rearranged in nondecreasing order
    for $k$ from 1 to $n-1$ do
        Insert $A[k]$ at its proper location within $A[0], A[1], \ldots, A[k]$.
```

Code Fragment 3.5: High-level description of the insertion-sort algorithm.

This is a simple, high-level description of insertion-sort. If we look back to Code Fragment 3.3 in Section 3.1.1, we see that the task of inserting a new entry into the list of high scores is almost identical to the task of inserting a newly considered element in insertion-sort (except that game scores were ordered from high to low). We provide a Java implementation of insertion-sort in Code Fragment 3.6, using an outer loop to consider each element in turn, and an inner loop that moves a newly considered element to its proper location relative to the (sorted) subarray of elements that are to its left. We illustrate an example run of the insertion-sort algorithm in Figure 3.5.

We note that if an array is already sorted, the inner loop of insertion-sort does only one comparison, determines that there is no swap needed, and returns back to the outer loop. Of course, we might have to do a lot more work than this if the input array is extremely out of order. In fact, we will have to do the most work if the input array is in decreasing order.

```
/** Insertion-sort of an array of characters into nondecreasing order */
public static void insertionSort(char[ ] data) {
  int n = data.length;
  for (int k = 1; k < n; k++) {                  // begin with second character
    char cur = data[k];                          // time to insert cur=data[k]
    int j = k;                                   // find correct index j for cur
    while (j > 0 && data[j−1] > cur) {           // thus, data[j-1] must go after cur
      data[j] = data[j−1];                       // slide data[j-1] rightward
      j−−;                                       // and consider previous j for cur
    }
    data[j] = cur;                               // this is the proper place for cur
  }
}
```

Code Fragment 3.6: Java code for performing insertion-sort on a character array.

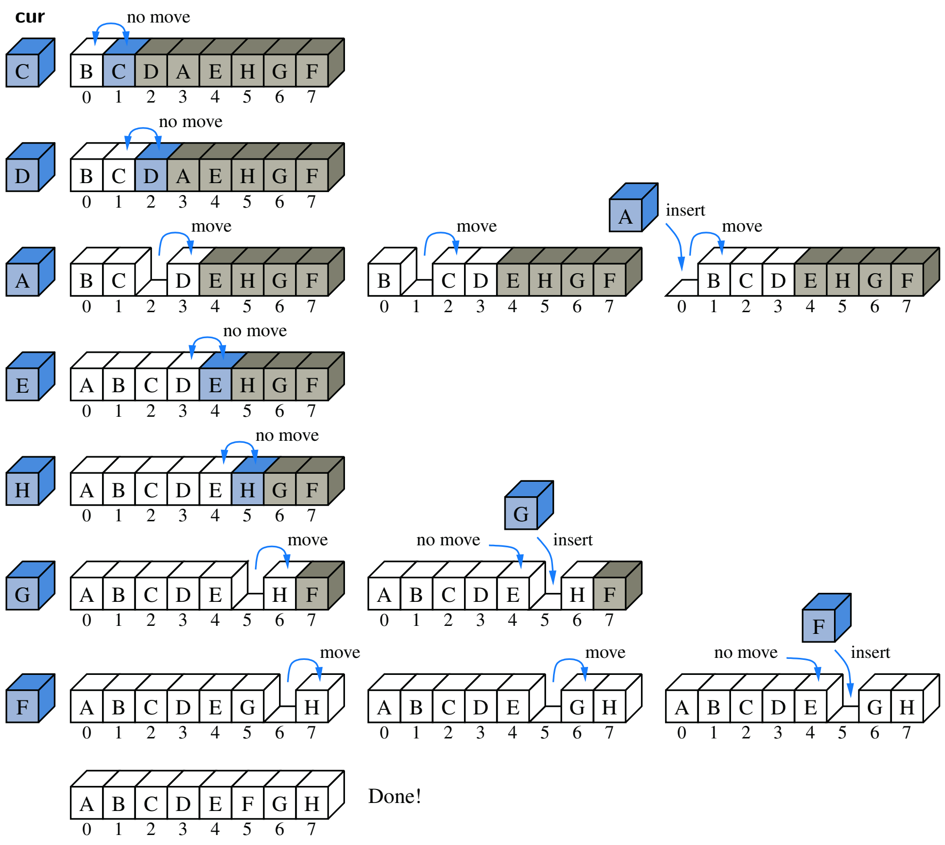


Figure 3.5: Execution of the insertion-sort algorithm on an array of eight characters. Each row corresponds to an iteration of the outer loop, and each copy of the sequence in a row corresponds to an iteration of the inner loop. The current element that is being inserted is highlighted in the array, and shown as the cur value.

3.1.3 java.util Methods for Arrays and Random Numbers

Because arrays are so important, Java provides a class, java.util.Arrays, with a number of built-in static methods for performing common tasks on arrays. Later in this book, we will describe the algorithms that several of these methods are based upon. For now, we provide an overview of the most commonly used methods of that class, as follows (more discussion is in Section 3.5.1):

equals(A, B):	Returns true if and only if the array A and the array B are equal. Two arrays are considered equal if they have the same number of elements and every corresponding pair of elements in the two arrays are equal. That is, A and B have the same values in the same order.
fill(A, x):	Stores value x in every cell of array A, provided the type of array A is defined so that it is allowed to store the value x.
copyOf(A, n):	Returns an array of size n such that the first k elements of this array are copied from A, where $k = \min\{n, A.\text{length}\}$. If $n > A.\text{length}$, then the last $n - A.\text{length}$ elements in this array will be padded with default values, e.g., 0 for an array of **int** and **null** for an array of objects.
copyOfRange(A, s, t):	Returns an array of size $t - s$ such that the elements of this array are copied in order from $A[s]$ to $A[t-1]$, where $s < t$, padded as with copyOf( ) if $t > A.\text{length}$.
toString(A):	Returns a String representation of the array A, beginning with [, ending with], and with elements of A displayed separated by string `", "`. The string representation of an element $A[i]$ is obtained using String.valueOf($A[i]$), which returns the string `"null"` for a **null** reference and otherwise calls $A[i]$.toString( ).
sort(A):	Sorts the array A based on a natural ordering of its elements, which must be comparable. Sorting algorithms are the focus of Chapter 12.
binarySearch(A, x):	Searches the *sorted* array A for value x, returning the index where it is found, or else the index of where it could be inserted while maintaining the sorted order. The binary-search algorithm is described in Section 5.1.3.

As static methods, these are invoked directly on the java.util.Arrays class, not on a particular instance of the class. For example, if data were an array, we could sort it with syntax, java.util.Arrays.sort(data), or with the shorter syntax Arrays.sort(data) if we first import the Arrays class (see Section 1.8).

PseudoRandom Number Generation

Another feature built into Java, which is often useful when testing programs dealing with arrays, is the ability to generate pseudorandom numbers, that is, numbers that appear to be random (but are not necessarily truly random). In particular, Java has a built-in class, java.util.Random, whose instances are ***pseudorandom number generators***, that is, objects that compute a sequence of numbers that are statistically random. These sequences are not actually random, however, in that it is possible to predict the next number in the sequence given the past list of numbers. Indeed, a popular pseudorandom number generator is to generate the next number, next, from the current number, cur, according to the formula (in Java syntax):

```
next = (a * cur + b) % n;
```

where a, b, and n are appropriately chosen integers, and % is the modulus operator. Something along these lines is, in fact, the method used by java.util.Random objects, with $n = 2^{48}$. It turns out that such a sequence can be proven to be statistically uniform, which is usually good enough for most applications requiring random numbers, such as games. For applications, such as computer security settings, where unpredictable random sequences are needed, this kind of formula should not be used. Instead, ideally a sample from a source that is actually random should be used, such as radio static coming from outer space.

Since the next number in a pseudorandom generator is determined by the previous number(s), such a generator always needs a place to start, which is called its ***seed***. The sequence of numbers generated for a given seed will always be the same. The seed for an instance of the java.util.Random class can be set in its constructor or with its setSeed( ) method.

One common trick to get a different sequence each time a program is run is to use a seed that will be different for each run. For example, we could use some timed input from a user or we could set the seed to the current time in milliseconds since January 1, 1970 (provided by method System.currentTimeMillis).

Methods of the java.util.Random class include the following:

nextBoolean( ):	Returns the next pseudorandom **boolean** value.
nextDouble( ):	Returns the next pseudorandom **double** value, between 0.0 and 1.0.
nextInt( ):	Returns the next pseudorandom **int** value.
nextInt(n):	Returns the next pseudorandom **int** value in the range from 0 up to but not including n.
setSeed(s):	Sets the seed of this pseudorandom number generator to the **long** s.

An Illustrative Example

We provide a short (but complete) illustrative program in Code Fragment 3.7.

```java
import java.util.Arrays;
import java.util.Random;
/** Program showing some array uses. */
public class ArrayTest {
  public static void main(String[ ] args) {
    int data[ ] = new int[10];
    Random rand = new Random( );              // a pseudo-random number generator
    rand.setSeed(System.currentTimeMillis( ));             // use current time as a seed
    // fill the data array with pseudo-random numbers from 0 to 99, inclusive
    for (int i = 0; i < data.length; i++)
      data[i] = rand.nextInt(100);                       // the next pseudo-random number
    int[ ] orig = Arrays.copyOf(data, data.length); // make a copy of the data array
    System.out.println("arrays equal before sort: "+Arrays.equals(data, orig));
    Arrays.sort(data);                          // sorting the data array (orig is unchanged)
    System.out.println("arrays equal after sort: " + Arrays.equals(data, orig));
    System.out.println("orig = " + Arrays.toString(orig));
    System.out.println("data = " + Arrays.toString(data));
  }
}
```

Code Fragment 3.7: A simple test of some built-in methods in java.util.Arrays.

We show a sample output of this program below:

```
arrays equal before sort: true
arrays equal after sort: false
orig = [41, 38, 48, 12, 28, 46, 33, 19, 10, 58]
data = [10, 12, 19, 28, 33, 38, 41, 46, 48, 58]
```

In another run, we got the following output:

```
arrays equal before sort: true
arrays equal after sort: false
orig = [87, 49, 70, 2, 59, 37, 63, 37, 95, 1]
data = [1, 2, 37, 37, 49, 59, 63, 70, 87, 95]
```

By using a pseudorandom number generator to determine program values, we get a different input to our program each time we run it. This feature is, in fact, what makes pseudorandom number generators useful for testing code, particularly when dealing with arrays. Even so, we should not use random test runs as a replacement for reasoning about our code, as we might miss important special cases in test runs. Note, for example, that there is a slight chance that the orig and data arrays will be equal even after data is sorted, namely, if orig is already ordered. The odds of this occurring are less than 1 in 3 million, so it's unlikely to happen during even a few thousand test runs; however, we need to reason that this is possible.

3.1.4 Simple Cryptography with Character Arrays

An important application of character arrays and strings is ***cryptography***, which is the science of secret messages. This field involves the process of ***encryption***, in which a message, called the ***plaintext***, is converted into a scrambled message, called the ***ciphertext***. Likewise, cryptography studies corresponding ways of performing ***decryption***, turning a ciphertext back into its original plaintext.

Arguably the earliest encryption scheme is the ***Caesar cipher***, which is named after Julius Caesar, who used this scheme to protect important military messages. (All of Caesar's messages were written in Latin, of course, which already makes them unreadable for most of us!) The Caesar cipher is a simple way to obscure a message written in a language that forms words with an alphabet.

The Caesar cipher involves replacing each letter in a message with the letter that is a certain number of letters after it in the alphabet. So, in an English message, we might replace each A with D, each B with E, each C with F, and so on, if shifting by three characters. We continue this approach all the way up to W, which is replaced with Z. Then, we let the substitution pattern ***wrap around***, so that we replace X with A, Y with B, and Z with C.

Converting Between Strings and Character Arrays

Given that strings are immutable, we cannot directly edit an instance to encrypt it. Instead, our goal will be to generate a new string. A convenient technique for performing string transformations is to create an equivalent array of characters, edit the array, and then reassemble a (new) string based on the array.

Java has support for conversions from strings to character arrays and vice versa. Given a string S, we can create a new character array matching S by using the method, S.toCharArray( ). For example, if s=`"bird"`, the method returns the character array A={`'b'`, `'i'`, `'r'`, `'d'`}. Conversely, there is a form of the String constructor that accepts a character array as a parameter. For example, with character array A={`'b'`, `'i'`, `'r'`, `'d'`}, the syntax **new** String(A) produces `"bird"`.

Using Character Arrays as Replacement Codes

If we were to number our letters like array indices, so that A is 0, B is 1, C is 2, then we can represent the replacement rule as a character array, encoder, such that A is mapped to encoder[0], B is mapped to encoder[1], and so on. Then, in order to find a replacement for a character in our Caesar cipher, we need to map the characters from A to Z to the respective numbers from 0 to 25. Fortunately, we can rely on the fact that characters are represented in Unicode by integer code points, and the code points for the uppercase letters of the Latin alphabet are consecutive (for simplicity, we restrict our encryption to uppercase letters).

Java allows us to “subtract” two characters from each other, with an integer result equal to their separation distance in the encoding. Given a variable c that is known to be an uppercase letter, the Java computation, $j =$ c − 'A' produces the desired index j. As a sanity check, if character c is 'A', then $j = 0$. When c is 'B', the difference is 1. In general, the integer j that results from such a calculation can be used as an index into our precomputed encoder array, as illustrated in Figure 3.6.

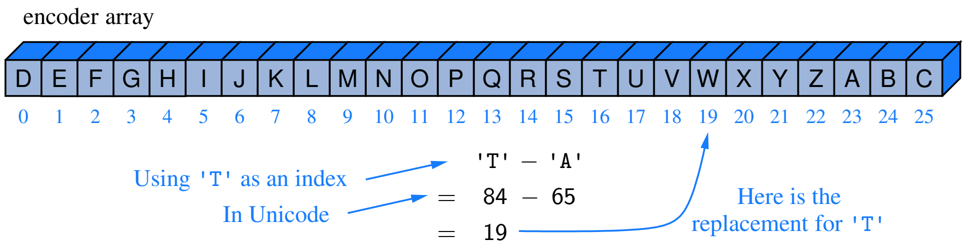


Figure 3.6: Illustrating the use of uppercase characters as indices, in this case to perform the replacement rule for Caesar cipher encryption.

The process of ***decrypting*** the message can be implemented by simply using a different character array to represent the replacement rule—one that effectively shifts characters in the opposite direction.

In Code Fragment 3.8, we present a Java class that performs the Caesar cipher with an arbitrary rotational shift. The constructor for the class builds the encoder and decoder translation arrays for the given rotation. We rely heavily on modular arithmetic, as a Caesar cipher with a rotation of r encodes the letter having index k with the letter having index $(k + r) \bmod 26$, where mod is the ***modulo*** operator, which returns the remainder after performing an integer division. This operator is denoted with % in Java, and it is exactly the operator we need to easily perform the wraparound at the end of the alphabet, for 26 mod 26 is 0, 27 mod 26 is 1, and 28 mod 26 is 2. The decoder array for the Caesar cipher is just the opposite—we replace each letter with the one r places before it, with wraparound; to avoid subtleties involving negative numbers and the modulus operator, we will replace the letter having code k with the letter having code $(k - r + 26) \bmod 26$.

With the encoder and decoder arrays in hand, the encryption and decryption algorithms are essentially the same, and so we perform both by means of a private utility method named transform. This method converts a string to a character array, performs the translation diagrammed in Figure 3.6 for any uppercase alphabet symbols, and finally returns a new string, constructed from the updated array.

The main method of the class, as a simple test, produces the following output:

```
Encryption code = DEFGHIJKLMNOPQRSTUVWXYZABC
Decryption code = XYZABCDEFGHIJKLMNOPQRSTUVW
Secret:  WKH HDJOH LV LQ SODB; PHHW DW MRH’V.
Message: THE EAGLE IS IN PLAY; MEET AT JOE’S.
```

```
/** Class for doing encryption and decryption using the Caesar Cipher. */
public class CaesarCipher {
  protected char[ ] encoder = new char[26];           // Encryption array
  protected char[ ] decoder = new char[26];           // Decryption array
  /** Constructor that initializes the encryption and decryption arrays */
  public CaesarCipher(int rotation) {
    for (int k=0; k < 26; k++) {
      encoder[k] = (char) ('A' + (k + rotation) % 26);
      decoder[k] = (char) ('A' + (k − rotation + 26) % 26);
    }
  }
  /** Returns String representing encrypted message. */
  public String encrypt(String message) {
    return transform(message, encoder);               // use encoder array
  }
  /** Returns decrypted message given encrypted secret. */
  public String decrypt(String secret) {
    return transform(secret, decoder);                // use decoder array
  }
  /** Returns transformation of original String using given code. */
  private String transform(String original, char[ ] code) {
    char[ ] msg = original.toCharArray( );
    for (int k=0; k < msg.length; k++)
      if (Character.isUpperCase(msg[k])) {            // we have a letter to change
        int j = msg[k] − 'A';                         // will be value from 0 to 25
        msg[k] = code[j];                             // replace the character
      }
    return new String(msg);
  }
  /** Simple main method for testing the Caesar cipher */
  public static void main(String[ ] args) {
    CaesarCipher cipher = new CaesarCipher(3);
    System.out.println("Encryption code = " + new String(cipher.encoder));
    System.out.println("Decryption code = " + new String(cipher.decoder));
    String message = "THE EAGLE IS IN PLAY; MEET AT JOE'S.";
    String coded = cipher.encrypt(message);
    System.out.println("Secret:  " + coded);
    String answer = cipher.decrypt(coded);
    System.out.println("Message: " + answer);         // should be plaintext again
  }
}
```

Code Fragment 3.8: A complete Java class for performing the Caesar cipher.

3.1.5 Two-Dimensional Arrays and Positional Games

Many computer games, be they strategy games, simulation games, or first-person conflict games, involve objects that reside in a two-dimensional space. Software for such ***positional games*** needs a way of representing objects in a two-dimensional space. A natural way to do this is with a ***two-dimensional array***, where we use two indices, say i and j, to refer to the cells in the array. The first index usually refers to a row number and the second to a column number. Given such an array, we can maintain two-dimensional game boards and perform other kinds of computations involving data stored in rows and columns.

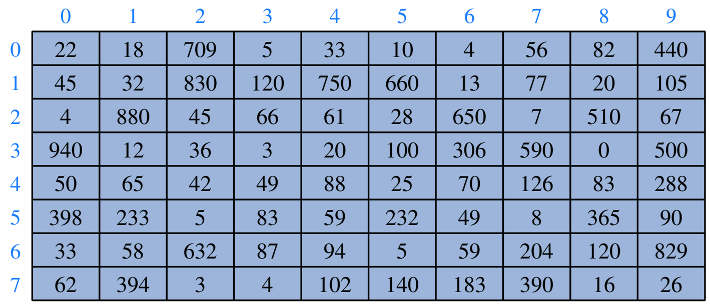
Arrays in Java are one-dimensional; we use a single index to access each cell of an array. Nevertheless, there is a way we can define two-dimensional arrays in Java—we can create a two-dimensional array as an array of arrays. That is, we can define a two-dimensional array to be an array with each of its cells being another array. Such a two-dimensional array is sometimes also called a ***matrix***. In Java, we may declare a two-dimensional array as follows:

```
int[ ][ ] data = new int[8][10];
```

This statement creates a two-dimensional "array of arrays," data, which is 8×10, having 8 rows and 10 columns. That is, data is an array of length 8 such that each element of data is an array of length 10 of integers. (See Figure 3.7.) The following would then be valid uses of array data and **int** variables i, j, and k:

```
data[i][i+1] = data[i][i] + 3;
j = data.length;              // j is 8
k = data[4].length;           // k is 10
```

Two-dimensional arrays have many applications to numerical analysis. Rather than going into the details of such applications, however, we explore an application of two-dimensional arrays for implementing a simple positional game.



	0	1	2	3	4	5	6	7	8	9
0	22	18	709	5	33	10	4	56	82	440
1	45	32	830	120	750	660	13	77	20	105
2	4	880	45	66	61	28	650	7	510	67
3	940	12	36	3	20	100	306	590	0	500
4	50	65	42	49	88	25	70	126	83	288
5	398	233	5	83	59	232	49	8	365	90
6	33	58	632	87	94	5	59	204	120	829
7	62	394	3	4	102	140	183	390	16	26

Figure 3.7: Illustration of a two-dimensional integer array, data, which has 8 rows and 10 columns. The value of data[3][5] is 100 and the value of data[6][2] is 632.

Tic-Tac-Toe

As most school children know, ***Tic-Tac-Toe*** is a game played in a three-by-three board. Two players—X and O—alternate in placing their respective marks in the cells of this board, starting with player X. If either player succeeds in getting three of his or her marks in a row, column, or diagonal, then that player wins.

This is admittedly not a sophisticated positional game, and it's not even that much fun to play, since a good player O can always force a tie. Tic-Tac-Toe's saving grace is that it is a nice, simple example showing how two-dimensional arrays can be used for positional games. Software for more sophisticated positional games, such as checkers, chess, or the popular simulation games, are all based on the same approach we illustrate here for using a two-dimensional array for Tic-Tac-Toe.

The basic idea is to use a two-dimensional array, board, to maintain the game board. Cells in this array store values that indicate if that cell is empty or stores an X or O. That is, board is a three-by-three matrix, whose middle row consists of the cells board[1][0], board[1][1], and board[1][2]. In our case, we choose to make the cells in the board array be integers, with a 0 indicating an empty cell, a 1 indicating an X, and a -1 indicating an O. This encoding allows us to have a simple way of testing if a given board configuration is a win for X or O, namely, if the values of a row, column, or diagonal add up to 3 or -3, respectively. We illustrate this approach in Figure 3.8.

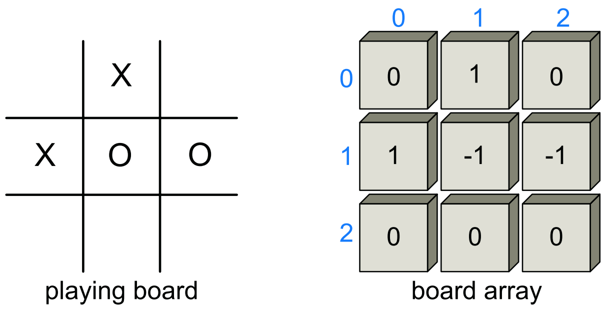


Figure 3.8: An illustration of a Tic-Tac-Toe board and the two-dimensional integer array, board, representing it.

We give a complete Java class for maintaining a Tic-Tac-Toe board for two players in Code Fragments 3.9 and 3.10. We show a sample output in Figure 3.9. Note that this code is just for maintaining the Tic-Tac-Toe board and registering moves; it doesn't perform any strategy or allow someone to play Tic-Tac-Toe against the computer. The details of such a program are beyond the scope of this chapter, but it might nonetheless make a good course project (see Exercise P-8.67).

```
/** Simulation of a Tic-Tac-Toe game (does not do strategy). */
public class TicTacToe {
  public static final int X = 1, O = −1;          // players
  public static final int EMPTY = 0;              // empty cell
  private int board[ ][ ] = new int[3][3];        // game board
  private int player;                             // current player
  /** Constructor */
  public TicTacToe( ) { clearBoard( ); }
  /** Clears the board */
  public void clearBoard( ) {
    for (int i = 0; i < 3; i++)
      for (int j = 0; j < 3; j++)
        board[i][j] = EMPTY;                      // every cell should be empty
    player = X;                                   // the first player is 'X'
  }
  /** Puts an X or O mark at position i,j. */
  public void putMark(int i, int j) throws IllegalArgumentException {
    if ((i < 0) || (i > 2) || (j < 0) || (j > 2))
      throw new IllegalArgumentException("Invalid board position");
    if (board[i][j] != EMPTY)
      throw new IllegalArgumentException("Board position occupied");
    board[i][j] = player;             // place the mark for the current player
    player = − player;                // switch players (uses fact that O = - X)
  }
  /** Checks whether the board configuration is a win for the given player. */
  public boolean isWin(int mark) {
    return ((board[0][0] + board[0][1] + board[0][2] == mark*3)       // row 0
             || (board[1][0] + board[1][1] + board[1][2] == mark*3)   // row 1
             || (board[2][0] + board[2][1] + board[2][2] == mark*3)   // row 2
             || (board[0][0] + board[1][0] + board[2][0] == mark*3)   // column 0
             || (board[0][1] + board[1][1] + board[2][1] == mark*3)   // column 1
             || (board[0][2] + board[1][2] + board[2][2] == mark*3)   // column 2
             || (board[0][0] + board[1][1] + board[2][2] == mark*3)   // diagonal
             || (board[2][0] + board[1][1] + board[0][2] == mark*3)); // rev diag
  }
  /** Returns the winning player's code, or 0 to indicate a tie (or unfinished game).*/
  public int winner( ) {
    if (isWin(X))
      return(X);
    else if (isWin(O))
      return(O);
    else
      return(0);
  }
```

Code Fragment 3.9: A simple, complete Java class for playing Tic-Tac-Toe between two players. (Continues in Code Fragment 3.10.)

```
  /** Returns a simple character string showing the current board. */
  public String toString( ) {
    StringBuilder sb = new StringBuilder( );
    for (int i=0; i<3; i++) {
      for (int j=0; j<3; j++) {
        switch (board[i][j]) {
        case X:          sb.append("X"); break;
        case O:          sb.append("O"); break;
        case EMPTY:      sb.append(" "); break;
        }
        if (j < 2) sb.append("|");                    // column boundary
      }
      if (i < 2) sb.append("\n-----\n");               // row boundary
    }
    return sb.toString( );
  }
  /** Test run of a simple game */
  public static void main(String[ ] args) {
    TicTacToe game = new TicTacToe( );
    /* X moves: */               /* O moves: */
    game.putMark(1,1);           game.putMark(0,2);
    game.putMark(2,2);           game.putMark(0,0);
    game.putMark(0,1);           game.putMark(2,1);
    game.putMark(1,2);           game.putMark(1,0);
    game.putMark(2,0);
    System.out.println(game);
    int winningPlayer = game.winner( );
    String[ ] outcome = {"O wins", "Tie", "X wins"};  // rely on ordering
    System.out.println(outcome[1 + winningPlayer]);
  }
}
```

Code Fragment 3.10: A simple, complete Java class for playing Tic-Tac-Toe between two players. (Continued from Code Fragment 3.9.)

```
O|X|O
-----
O|X|X
-----
X|O|X
Tie
```

Figure 3.9: Sample output of a Tic-Tac-Toe game.

3.2 Singly Linked Lists

In the previous section, we presented the array data structure and discussed some of its applications. Arrays are great for storing things in a certain order, but they have drawbacks. The capacity of the array must be fixed when it is created, and insertions and deletions at interior positions of an array can be time consuming if many elements must be shifted.

In this section, we introduce a data structure known as a ***linked list***, which provides an alternative to an array-based structure. A linked list, in its simplest form, is a collection of ***nodes*** that collectively form a linear sequence. In a ***singly linked list***, each node stores a reference to an object that is an element of the sequence, as well as a reference to the next node of the list (see Figure 3.10).

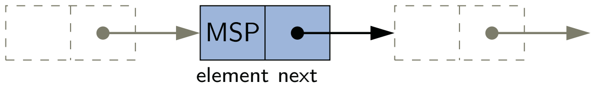


Figure 3.10: Example of a node instance that forms part of a singly linked list. The node's element field refers to an object that is an element of the sequence (the airport code MSP, in this example), while the next field refers to the subsequent node of the linked list (or null if there is no further node).

A linked list's representation relies on the collaboration of many objects (see Figure 3.11). Minimally, the linked list instance must keep a reference to the first node of the list, known as the ***head***. Without an explicit reference to the head, there would be no way to locate that node (or indirectly, any others). The last node of the list is known as the ***tail***. The tail of a list can be found by ***traversing*** the linked list— starting at the head and moving from one node to another by following each node's next reference. We can identify the tail as the node having **null** as its next reference. This process is also known as ***link hopping*** or ***pointer hopping***. However, storing an explicit reference to the tail node is a common efficiency to avoid such a traversal. In similar regard, it is common for a linked list instance to keep a count of the total number of nodes that comprise the list (also known as the ***size*** of the list), to avoid traversing the list to count the nodes.

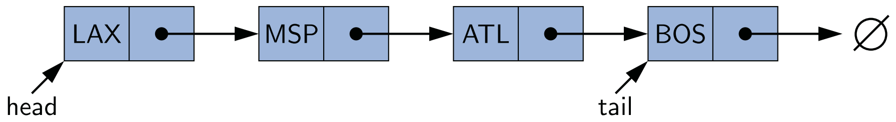


Figure 3.11: Example of a singly linked list whose elements are strings indicating airport codes. The list instance maintains a member named head that refers to the first node of the list, and another member named tail that refers to the last node of the list. The **null** value is denoted as Ø.

Inserting an Element at the Head of a Singly Linked List

An important property of a linked list is that it does not have a predetermined fixed size; it uses space proportional to its current number of elements. When using a singly linked list, we can easily insert an element at the head of the list, as shown in Figure 3.12, and described with pseudocode in Code Fragment 3.11. The main idea is that we create a new node, set its element to the new element, set its next link to refer to the current head, and set the list's head to point to the new node.

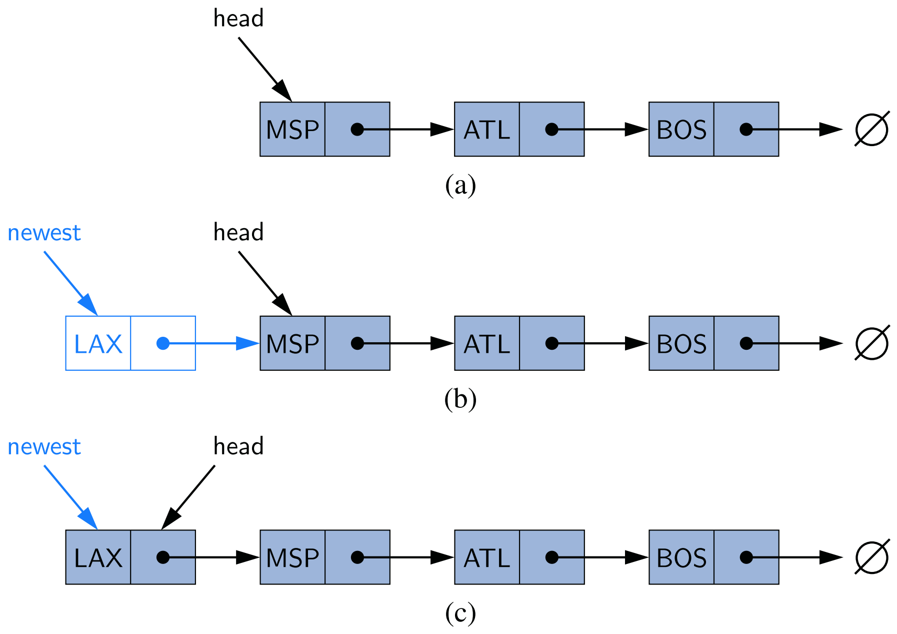


Figure 3.12: Insertion of an element at the head of a singly linked list: (a) before the insertion; (b) after a new node is created and linked to the existing head; (c) after reassignment of the head reference to the newest node.

```
Algorithm addFirst(e):
    newest = Node(e)   {create new node instance storing reference to element e}
    newest.next = head   {set new node's next to reference the old head node}
    head = newest   {set variable head to reference the new node}
    size = size + 1   {increment the node count}
```

Code Fragment 3.11: Inserting a new element at the beginning of a singly linked list. Note that we set the next pointer of the new node ***before*** we reassign variable head to it. If the list were initially empty (i.e., head is null), then a natural consequence is that the new node has its next reference set to null.

Inserting an Element at the Tail of a Singly Linked List

We can also easily insert an element at the tail of the list, provided we keep a reference to the tail node, as shown in Figure 3.13. In this case, we create a new node, assign its next reference to null, set the next reference of the tail to point to this new node, and then update the tail reference itself to this new node. We give pseudocode for the process in Code Fragment 3.12.

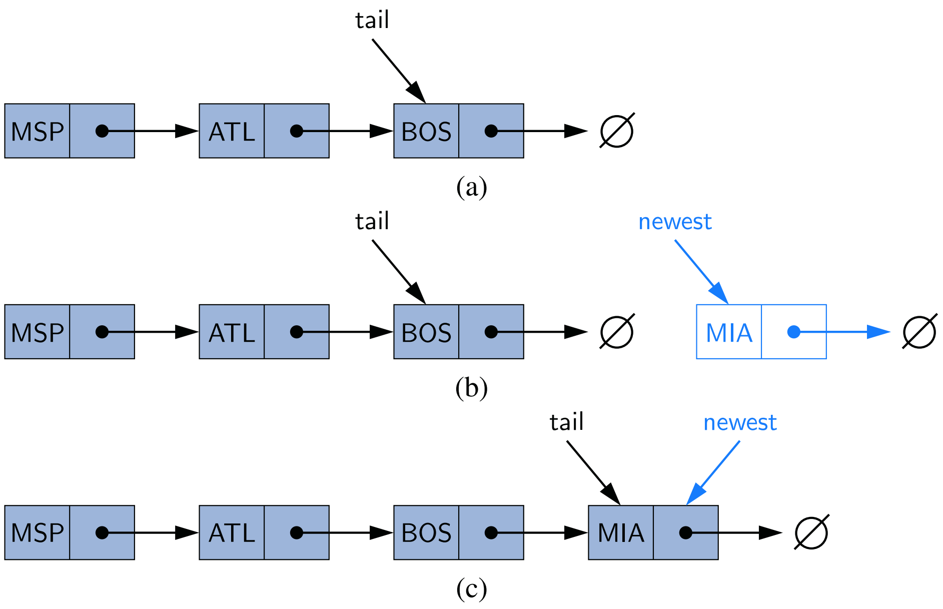


Figure 3.13: Insertion at the tail of a singly linked list: (a) before the insertion; (b) after creation of a new node; (c) after reassignment of the tail reference. Note that we must set the next link of the tail node in (b) before we assign the tail variable to point to the new node in (c).

```
Algorithm addLast(e):
    newest = Node(e)    {create new node instance storing reference to element e}
    newest.next = null           {set new node's next to reference the null object}
    tail.next = newest                    {make old tail node point to new node}
    tail = newest                       {set variable tail to reference the new node}
    size = size + 1                                       {increment the node count}
```

Code Fragment 3.12: Inserting a new node at the end of a singly linked list. Note that we set the next pointer for the old tail node ***before*** we make variable tail point to the new node. This code would need to be adjusted for inserting onto an empty list, since there would not be an existing tail node.

Removing an Element from a Singly Linked List

Removing an element from the ***head*** of a singly linked list is essentially the reverse operation of inserting a new element at the head. This operation is illustrated in Figure 3.14 and described in detail in Code Fragment 3.13.

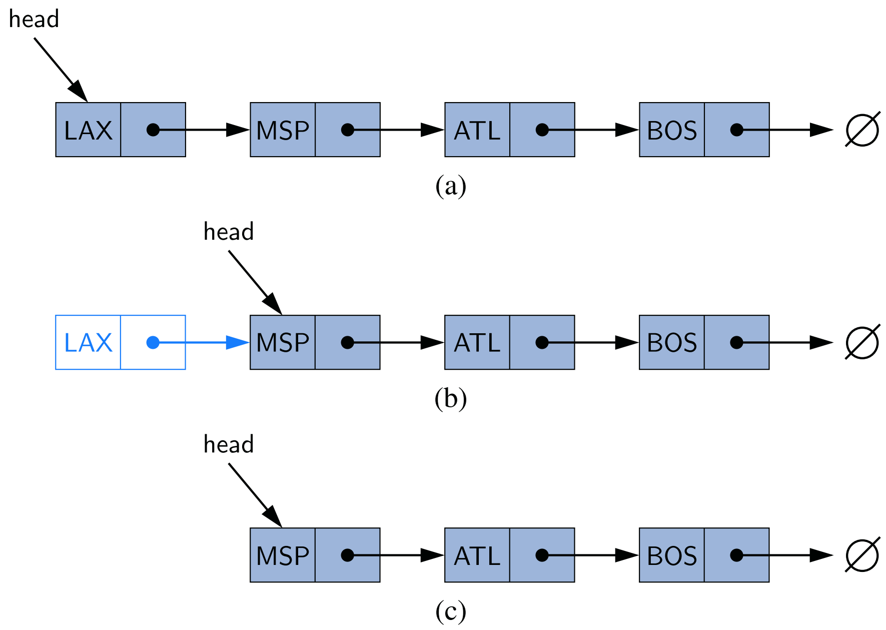


Figure 3.14: Removal of an element at the head of a singly linked list: (a) before the removal; (b) after "linking out" the old head; (c) final configuration.

```
Algorithm removeFirst( ):
    if head == null then
       the list is empty.
    head = head.next                  {make head point to next node (or null)}
    size = size − 1                          {decrement the node count}
```

Code Fragment 3.13: Removing the node at the beginning of a singly linked list.

Unfortunately, we cannot easily delete the last node of a singly linked list. Even if we maintain a tail reference directly to the last node of the list, we must be able to access the node ***before*** the last node in order to remove the last node. But we cannot reach the node before the tail by following next links from the tail. The only way to access this node is to start from the head of the list and search all the way through the list. But such a sequence of link-hopping operations could take a long time. If we want to support such an operation efficiently, we will need to make our list ***doubly linked*** (as we do in Section 3.4).

3.2.1 Implementing a Singly Linked List Class

In this section, we present a complete implementation of a SinglyLinkedList class, supporting the following methods:

size():	Returns the number of elements in the list.
isEmpty():	Returns **true** if the list is empty, and **false** otherwise.
first():	Returns (but does not remove) the first element in the list.
last():	Returns (but does not remove) the last element in the list.
addFirst(e):	Adds a new element to the front of the list.
addLast(e):	Adds a new element to the end of the list.
removeFirst():	Removes and returns the first element of the list.

If first(), last(), or removeFirst() are called on a list that is empty, we will simply return a **null** reference and leave the list unchanged.

Because it does not matter to us what type of elements are stored in the list, we use Java's ***generics framework*** (see Section 2.5.2) to define our class with a formal type parameter E that represents the user's desired element type.

Our implementation also takes advantage of Java's support for ***nested classes*** (see Section 2.6), as we define a private Node class within the scope of the public SinglyLinkedList class. Code Fragment 3.14 presents the Node class definition, and Code Fragment 3.15 the rest of the SinglyLinkedList class. Having Node as a nested class provides strong encapsulation, shielding users of our class from the underlying details about nodes and links. This design also allows Java to differentiate this node type from forms of nodes we may define for use in other structures.

```
public class SinglyLinkedList<E> {
  //---------------- nested Node class ----------------
  private static class Node<E> {
    private E element;              // reference to the element stored at this node
    private Node<E> next;           // reference to the subsequent node in the list
    public Node(E e, Node<E> n) {
      element = e;
      next = n;
    }
    public E getElement() { return element; }
    public Node<E> getNext() { return next; }
    public void setNext(Node<E> n) { next = n; }
  } //----------- end of nested Node class -----------
  ... rest of SinglyLinkedList class will follow ...
```

Code Fragment 3.14: A nested Node class within the SinglyLinkedList class. (The remainder of the SinglyLinkedList class will be given in Code Fragment 3.15.)

```
public class SinglyLinkedList<E> {
  (nested Node class goes here)
  // instance variables of the SinglyLinkedList
  private Node<E> head = null;              // head node of the list (or null if empty)
  private Node<E> tail = null;              // last node of the list (or null if empty)
  private int size = 0;                     // number of nodes in the list
  public SinglyLinkedList( ) { }            // constructs an initially empty list
  // access methods
  public int size( ) { return size; }
  public boolean isEmpty( ) { return size == 0; }
  public E first( ) {                 // returns (but does not remove) the first element
    if (isEmpty( )) return null;
    return head.getElement( );
  }
  public E last( ) {                  // returns (but does not remove) the last element
    if (isEmpty( )) return null;
    return tail.getElement( );
  }
  // update methods
  public void addFirst(E e) {               // adds element e to the front of the list
    head = new Node<>(e, head);             // create and link a new node
    if (size == 0)
      tail = head;                          // special case: new node becomes tail also
    size++;
  }
  public void addLast(E e) {                // adds element e to the end of the list
    Node<E> newest = new Node<>(e, null);     // node will eventually be the tail
    if (isEmpty( ))
      head = newest;                        // special case: previously empty list
    else
      tail.setNext(newest);                 // new node after existing tail
    tail = newest;                          // new node becomes the tail
    size++;
  }
  public E removeFirst( ) {                 // removes and returns the first element
    if (isEmpty( )) return null;            // nothing to remove
    E answer = head.getElement( );
    head = head.getNext( );                 // will become null if list had only one node
    size--;
    if (size == 0)
      tail = null;                          // special case as list is now empty
    return answer;
  }
}
```

Code Fragment 3.15: The SinglyLinkedList class definition (when combined with the nested Node class of Code Fragment 3.14).

3.3 Circularly Linked Lists

Linked lists are traditionally viewed as storing a sequence of items in a linear order, from first to last. However, there are many applications in which data can be more naturally viewed as having a ***cyclic order***, with well-defined neighboring relationships, but no fixed beginning or end.

For example, many multiplayer games are turn-based, with player A taking a turn, then player B, then player C, and so on, but eventually back to player A again, and player B again, with the pattern repeating. As another example, city buses and subways often run on a continuous loop, making stops in a scheduled order, but with no designated first or last stop per se. We next consider another important example of a cyclic order in the context of computer operating systems.

3.3.1 Round-Robin Scheduling

One of the most important roles of an operating system is in managing the many processes that are currently active on a computer, including the scheduling of those processes on one or more central processing units (CPUs). In order to support the responsiveness of an arbitrary number of concurrent processes, most operating systems allow processes to effectively share use of the CPUs, using some form of an algorithm known as ***round-robin scheduling***. A process is given a short turn to execute, known as a ***time slice***, but it is interrupted when the slice ends, even if its job is not yet complete. Each active process is given its own time slice, taking turns in a cyclic order. New processes can be added to the system, and processes that complete their work can be removed.

A round-robin scheduler could be implemented with a traditional linked list, by repeatedly performing the following steps on linked list L (see Figure 3.15):

1. process $p = L.\mathsf{removeFirst}(\,)$
2. Give a time slice to process p
3. $L.\mathsf{addLast}(p)$

Unfortunately, there are drawbacks to the use of a traditional linked list for this purpose. It is unnecessarily inefficient to repeatedly throw away a node from one end of the list, only to create a new node for the same element when reinserting it, not to mention the various updates that are performed to decrement and increment the list's size and to unlink and relink nodes.

In the remainder of this section, we demonstrate how a slight modification to our singly linked list implementation can be used to provide a more efficient data structure for representing a cyclic order.

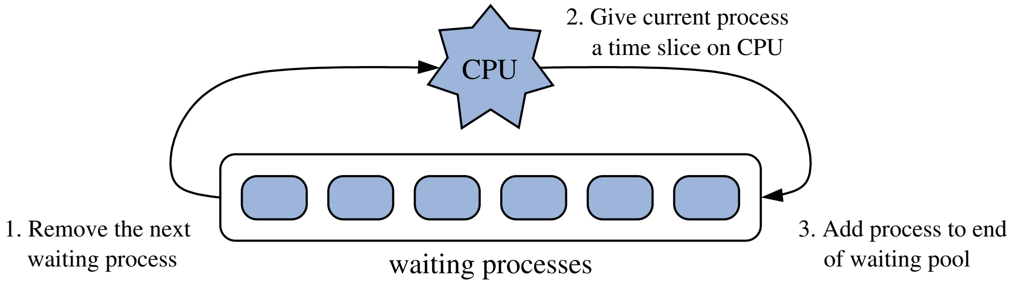

CPU
2. Give current process a time slice on CPU
1. Remove the next waiting process
waiting processes
3. Add process to end of waiting pool

Figure 3.15: The three iterative steps for round-robin scheduling.

3.3.2 Designing and Implementing a Circularly Linked List

In this section, we design a structure known as a ***circularly linked list***, which is essentially a singularly linked list in which the next reference of the tail node is set to refer back to the head of the list (rather than **null**), as shown in Figure 3.16.

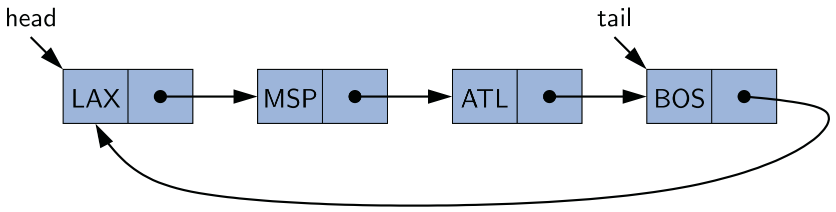


Figure 3.16: Example of a singly linked list with circular structure.

We use this model to design and implement a new CircularlyLinkedList class, which supports all of the public behaviors of our SinglyLinkedList class and one additional update method:

rotate( ): Moves the first element to the end of the list.

With this new operation, round-robin scheduling can be efficiently implemented by repeatedly performing the following steps on a circularly linked list C:

1. Give a time slice to process C.first( )
2. C.rotate( )

Additional Optimization

In implementing a new class, we make one additional optimization—we no longer explicitly maintain the head reference. So long as we maintain a reference to the tail, we can locate the head as tail.getNext( ). Maintaining only the tail reference not only saves a bit on memory usage, it makes the code simpler and more efficient, as it removes the need to perform additional operations to keep a head reference current. In fact, our new implementation is arguably superior to our original singly linked list implementation, even if we are not interested in the new rotate method.

Operations on a Circularly Linked List

Implementing the new rotate method is quite trivial. We do not move any nodes or elements, we simply advance the tail reference to point to the node that follows it (the implicit head of the list). Figure 3.17 illustrates this operation using a more symmetric visualization of a circularly linked list.

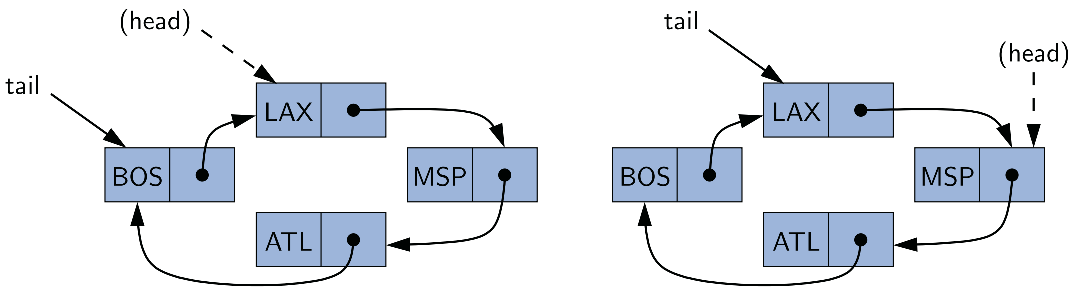


Figure 3.17: The rotation operation on a circularly linked list: (a) before the rotation, representing sequence { LAX, MSP, ATL, BOS }; (b) after the rotation, representing sequence { MSP, ATL, BOS, LAX }. We display the implicit head reference, which is identified only as tail.getNext( ) within the implementation.

We can add a new element at the front of the list by creating a new node and linking it just *after* the tail of the list, as shown in Figure 3.18. To implement the addLast method, we can rely on the use of a call to addFirst and then immediately advance the tail reference so that the newest node becomes the last.

Removing the first node from a circularly linked list can be accomplished by simply updating the next field of the tail node to bypass the implicit head. A Java implementation of all methods of the CircularlyLinkedList class is given in Code Fragment 3.16.

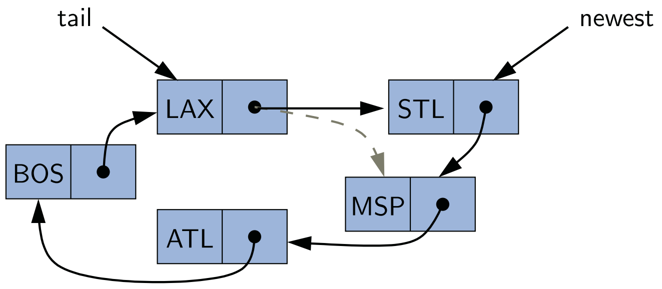


Figure 3.18: Effect of a call to addFirst(STL) on the circularly linked list of Figure 3.17(b). The variable newest has local scope during the execution of the method. Notice that when the operation is complete, STL is the first element of the list, as it is stored within the implicit head, tail.getNext( ).

```
public class CircularlyLinkedList<E> {
  (nested node class identical to that of the SinglyLinkedList class)
  // instance variables of the CircularlyLinkedList
  private Node<E> tail = null;                  // we store tail (but not head)
  private int size = 0;                         // number of nodes in the list
  public CircularlyLinkedList( ) { }            // constructs an initially empty list
  // access methods
  public int size( ) { return size; }
  public boolean isEmpty( ) { return size == 0; }
  public E first( ) {                  // returns (but does not remove) the first element
    if (isEmpty( )) return null;
    return tail.getNext( ).getElement( );       // the head is *after* the tail
  }
  public E last( ) {                   // returns (but does not remove) the last element
    if (isEmpty( )) return null;
    return tail.getElement( );
  }
  // update methods
  public void rotate( ) {              // rotate the first element to the back of the list
    if (tail != null)                           // if empty, do nothing
      tail = tail.getNext( );                   // the old head becomes the new tail
  }
  public void addFirst(E e) {                   // adds element e to the front of the list
    if (size == 0) {
      tail = new Node<>(e, null);
      tail.setNext(tail);                       // link to itself circularly
    } else {
      Node<E> newest = new Node<>(e, tail.getNext( ));
      tail.setNext(newest);
    }
    size++;
  }
  public void addLast(E e) {                    // adds element e to the end of the list
    addFirst(e);                                // insert new element at front of list
    tail = tail.getNext( );                     // now new element becomes the tail
  }
  public E removeFirst( ) {                     // removes and returns the first element
    if (isEmpty( )) return null;                // nothing to remove
    Node<E> head = tail.getNext( );
    if (head == tail) tail = null;              // must be the only node left
    else tail.setNext(head.getNext( ));         // removes "head" from the list
    size−−;
    return head.getElement( );
  }
}
```

Code Fragment 3.16: Implementation of the CircularlyLinkedList class.

3.4 Doubly Linked Lists

In a singly linked list, each node maintains a reference to the node that is immediately after it. We have demonstrated the usefulness of such a representation when managing a sequence of elements. However, there are limitations that stem from the asymmetry of a singly linked list. In Section 3.2, we demonstrated that we can efficiently insert a node at either end of a singly linked list, and can delete a node at the head of a list, but we are unable to efficiently delete a node at the tail of the list. More generally, we cannot efficiently delete an arbitrary node from an interior position of the list if only given a reference to that node, because we cannot determine the node that immediately *precedes* the node to be deleted (yet, that node needs to have its next reference updated).

To provide greater symmetry, we define a linked list in which each node keeps an explicit reference to the node before it and a reference to the node after it. Such a structure is known as a ***doubly linked list***. These lists allow a greater variety of $O(1)$-time update operations, including insertions and deletions at arbitrary positions within the list. We continue to use the term "next" for the reference to the node that follows another, and we introduce the term "prev" for the reference to the node that precedes it.

Header and Trailer Sentinels

In order to avoid some special cases when operating near the boundaries of a doubly linked list, it helps to add special nodes at both ends of the list: a ***header*** node at the beginning of the list, and a ***trailer*** node at the end of the list. These "dummy" nodes are known as ***sentinels*** (or guards), and they do not store elements of the primary sequence. A doubly linked list with such sentinels is shown in Figure 3.19.

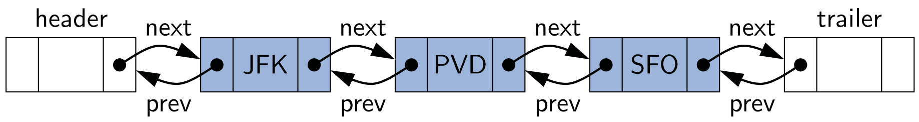


Figure 3.19: A doubly linked list representing the sequence { JFK, PVD, SFO }, using sentinels header and trailer to demarcate the ends of the list.

When using sentinel nodes, an empty list is initialized so that the next field of the header points to the trailer, and the prev field of the trailer points to the header; the remaining fields of the sentinels are irrelevant (presumably null, in Java). For a nonempty list, the header's next will refer to a node containing the first real element of a sequence, just as the trailer's prev references the node containing the last element of a sequence.

Advantage of Using Sentinels

Although we could implement a doubly linked list without sentinel nodes (as we did with our singly linked list in Section 3.2), the slight extra memory devoted to the sentinels greatly simplifies the logic of our operations. Most notably, the header and trailer nodes never change—only the nodes between them change. Furthermore, we can treat all insertions in a unified manner, because a new node will always be placed between a pair of existing nodes. In similar fashion, every element that is to be deleted is guaranteed to be stored in a node that has neighbors on each side.

For contrast, we look at our SinglyLinkedList implementation from Section 3.2. Its addLast method required a conditional (lines 39–42 of Code Fragment 3.15) to manage the special case of inserting into an empty list. In the general case, the new node was linked after the existing tail. But when adding to an empty list, there is no existing tail; instead it is necessary to reassign head to reference the new node. The use of a sentinel node in that implementation would eliminate the special case, as there would always be an existing node (possibly the header) before a new node.

Inserting and Deleting with a Doubly Linked List

Every insertion into our doubly linked list representation will take place between a pair of existing nodes, as diagrammed in Figure 3.20. For example, when a new element is inserted at the front of the sequence, we will simply add the new node *between* the header and the node that is currently after the header. (See Figure 3.21.)

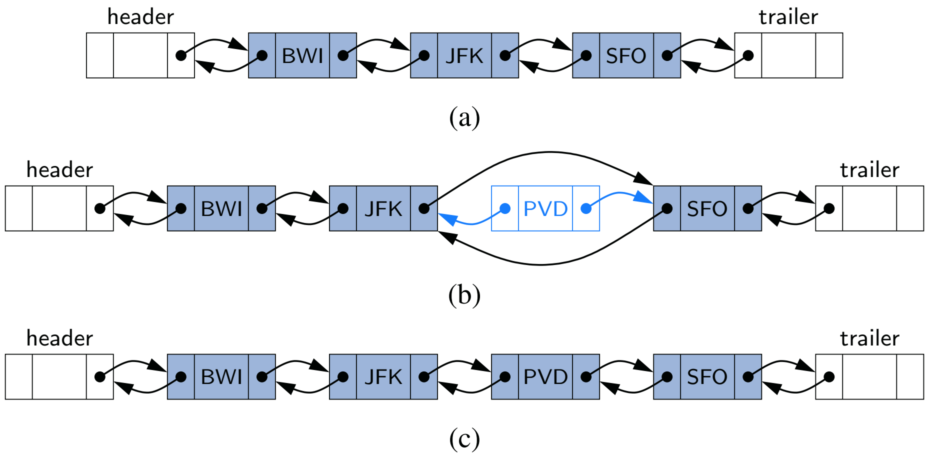


Figure 3.20: Adding an element to a doubly linked list with header and trailer sentinels: (a) before the operation; (b) after creating the new node; (c) after linking the neighbors to the new node.

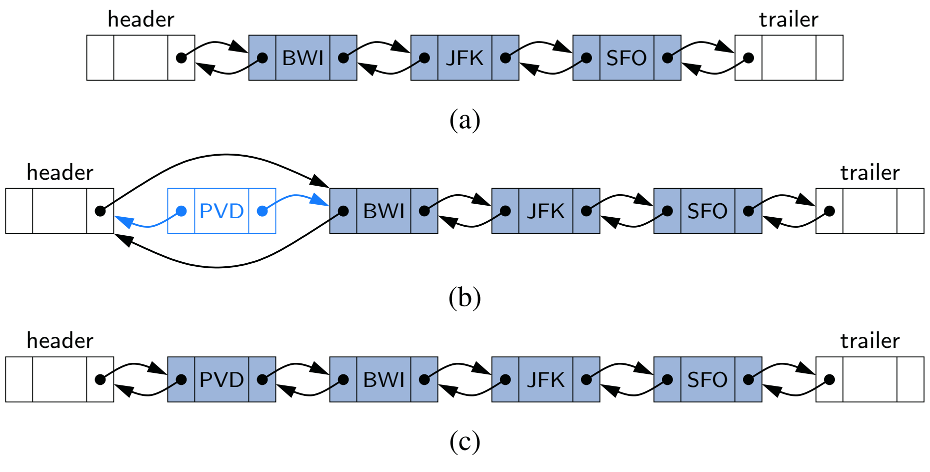


Figure 3.21: Adding an element to the front of a sequence represented by a doubly linked list with header and trailer sentinels: (a) before the operation; (b) after creating the new node; (c) after linking the neighbors to the new node.

The deletion of a node, portrayed in Figure 3.22, proceeds in the opposite fashion of an insertion. The two neighbors of the node to be deleted are linked directly to each other, thereby bypassing the original node. As a result, that node will no longer be considered part of the list and it can be reclaimed by the system. Because of our use of sentinels, the same implementation can be used when deleting the first or the last element of a sequence, because even such an element will be stored at a node that lies between two others.

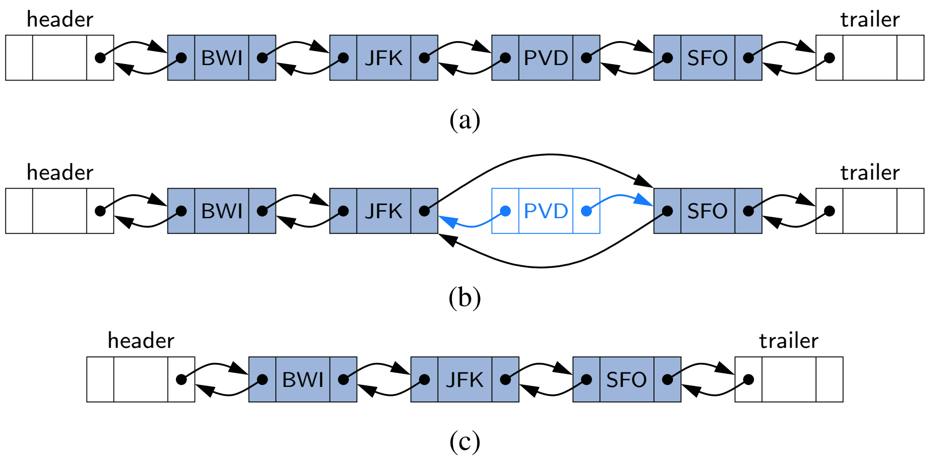


Figure 3.22: Removing the element PVD from a doubly linked list: (a) before the removal; (b) after linking out the old node; (c) after the removal (and garbage collection).

3.4.1 Implementing a Doubly Linked List Class

In this section, we present a complete implementation of a DoublyLinkedList class, supporting the following public methods:

size( ):	Returns the number of elements in the list.
isEmpty( ):	Returns **true** if the list is empty, and **false** otherwise.
first( ):	Returns (but does not remove) the first element in the list.
last( ):	Returns (but does not remove) the last element in the list.
addFirst(*e*):	Adds a new element to the front of the list.
addLast(*e*):	Adds a new element to the end of the list.
removeFirst( ):	Removes and returns the first element of the list.
removeLast( ):	Removes and returns the last element of the list.

If first( ), last( ), removeFirst( ), or removeLast( ) are called on a list that is empty, we will return a **null** reference and leave the list unchanged.

Although we have seen that it is possible to add or remove an element at an internal position of a doubly linked list, doing so requires knowledge of one or more nodes, to identify the position at which the operation should occur. In this chapter, we prefer to maintain encapsulation, with a private, nested Node class. In Chapter 7, we will revisit the use of doubly linked lists, offering a more advanced interface that supports internal insertions and deletions while maintaining encapsulation.

Code Fragments 3.17 and 3.18 present the DoublyLinkedList class implementation. As we did with our SinglyLinkedList class, we use the generics framework to accept any type of element. The nested Node class for the doubly linked list is similar to that of the singly linked list, except with support for an additional prev reference to the preceding node.

Our use of sentinel nodes, header and trailer, impacts the implementation in several ways. We create and link the sentinels when constructing an empty list (lines 25–29). We also keep in mind that the first element of a nonempty list is stored in the node just *after* the header (not in the header itself), and similarly that the last element is stored in the node just *before* the trailer.

The sentinels greatly ease our implementation of the various update methods. We will provide a private method, addBetween, to handle the general case of an insertion, and then we will rely on that utility as a straightforward method to implement both addFirst and addLast. In similar fashion, we will define a private remove method that can be used to easily implement both removeFirst and removeLast.

```
/** A basic doubly linked list implementation. */
public class DoublyLinkedList<E> {
  //---------------- nested Node class ----------------
  private static class Node<E> {
    private E element;               // reference to the element stored at this node
    private Node<E> prev;            // reference to the previous node in the list
    private Node<E> next;            // reference to the subsequent node in the list
    public Node(E e, Node<E> p, Node<E> n) {
      element = e;
      prev = p;
      next = n;
    }
    public E getElement() { return element; }
    public Node<E> getPrev() { return prev; }
    public Node<E> getNext() { return next; }
    public void setPrev(Node<E> p) { prev = p; }
    public void setNext(Node<E> n) { next = n; }
  } //----------- end of nested Node class -----------

  // instance variables of the DoublyLinkedList
  private Node<E> header;                          // header sentinel
  private Node<E> trailer;                         // trailer sentinel
  private int size = 0;                            // number of elements in the list
  /** Constructs a new empty list. */
  public DoublyLinkedList() {
    header = new Node<>(null, null, null);         // create header
    trailer = new Node<>(null, header, null);      // trailer is preceded by header
    header.setNext(trailer);                       // header is followed by trailer
  }
  /** Returns the number of elements in the linked list. */
  public int size() { return size; }
  /** Tests whether the linked list is empty. */
  public boolean isEmpty() { return size == 0; }
  /** Returns (but does not remove) the first element of the list. */
  public E first() {
    if (isEmpty()) return null;
    return header.getNext().getElement();          // first element is beyond header
  }
  /** Returns (but does not remove) the last element of the list. */
  public E last() {
    if (isEmpty()) return null;
    return trailer.getPrev().getElement();         // last element is before trailer
  }
```

Code Fragment 3.17: Implementation of the DoublyLinkedList class. (Continues in Code Fragment 3.18.)

```
  // public update methods
  /** Adds element e to the front of the list. */
  public void addFirst(E e) {
    addBetween(e, header, header.getNext());          // place just after the header
  }
  /** Adds element e to the end of the list. */
  public void addLast(E e) {
    addBetween(e, trailer.getPrev(), trailer);         // place just before the trailer
  }
  /** Removes and returns the first element of the list. */
  public E removeFirst() {
    if (isEmpty()) return null;                       // nothing to remove
    return remove(header.getNext());                  // first element is beyond header
  }
  /** Removes and returns the last element of the list. */
  public E removeLast() {
    if (isEmpty()) return null;                       // nothing to remove
    return remove(trailer.getPrev());                 // last element is before trailer
  }

  // private update methods
  /** Adds element e to the linked list in between the given nodes. */
  private void addBetween(E e, Node<E> predecessor, Node<E> successor) {
    // create and link a new node
    Node<E> newest = new Node<>(e, predecessor, successor);
    predecessor.setNext(newest);
    successor.setPrev(newest);
    size++;
  }
  /** Removes the given node from the list and returns its element. */
  private E remove(Node<E> node) {
    Node<E> predecessor = node.getPrev();
    Node<E> successor = node.getNext();
    predecessor.setNext(successor);
    successor.setPrev(predecessor);
    size--;
    return node.getElement();
  }
} //----------- end of DoublyLinkedList class -----------
```

Code Fragment 3.18: Implementation of the public and private update methods for the DoublyLinkedList class. (Continued from Code Fragment 3.17.)

3.5 Equivalence Testing

When working with reference types, there are many different notions of what it means for one expression to be equal to another. At the lowest level, if a and b are reference variables, then expression a == b tests whether a and b refer to the same object (or if both are set to the **null** value).

However, for many types there is a higher-level notion of two variables being considered "equivalent" even if they do not actually refer to the same instance of the class. For example, we typically want to consider two String instances to be equivalent to each other if they represent the identical sequence of characters.

To support a broader notion of equivalence, all object types support a method named **equals**. Users of reference types should rely on the syntax a.equals(b), unless they have a specific need to test the more narrow notion of identity. The equals method is formally defined in the Object class, which serves as a superclass for all reference types, but that implementation reverts to returning the value of expression a == b. Defining a more meaningful notion of equivalence requires knowledge about a class and its representation.

The author of each class has a responsibility to provide an implementation of the equals method, which overrides the one inherited from Object, if there is a more relevant definition for the equivalence of two instances. For example, Java's String class redefines equals to test character-for-character equivalence.

Great care must be taken when overriding the notion of equality, as the consistency of Java's libraries depends upon the equals method defining what is known as an ***equivalence relation*** in mathematics, satisfying the following properties:

Treatment of null: For any nonnull reference variable x, the call x.equals(**null**) should return **false** (that is, nothing equals **null** except **null**).

Reflexivity: For any nonnull reference variable x, the call x.equals(x) should return **true** (that is, an object should equal itself).

Symmetry: For any nonnull reference variables x and y, the calls x.equals(y) and y.equals(x) should return the same value.

Transitivity: For any nonnull reference variables x, y, and z, if both calls x.equals(y) and y.equals(z) return **true**, then call x.equals(z) must return **true** as well.

While these properties may seem intuitive, it can be challenging to properly implement equals for some data structures, especially in an object-oriented context, with inheritance and generics. For most of the data structures in this book, we omit the implementation of a valid equals method (leaving it as an exercise). However, in this section, we consider the treatment of equivalence testing for both arrays and linked lists, including a concrete example of a proper implementation of the equals method for our SinglyLinkedList class.

3.5.1 Equivalence Testing with Arrays

As we mentioned in Section 1.3, arrays are a reference type in Java, but not technically a class. However, the java.util.Arrays class, introduced in Section 3.1.3, provides additional static methods that are useful when processing arrays. The following provides a summary of the treatment of equivalence for arrays, assuming that variables a and b refer to array objects:

a == b:	Tests if a and b refer to the same underlying array instance.
a.equals(b):	Interestingly, this is identical to a == b. Arrays are not a true class type and do not override the Object.equals method.
Arrays.equals(a,b):	This provides a more intuitive notion of equivalence, returning **true** if the arrays have the same length and all pairs of corresponding elements are "equal" to each other. More specifically, if the array elements are primitives, then it uses the standard == to compare values. If elements of the arrays are a reference type, then it makes pairwise comparisons a[k].equals(b[k]) in evaluating the equivalence.

For most applications, the Arrays.equals behavior captures the appropriate notion of equivalence. However, there is an additional complication when using multidimensional arrays. The fact that two-dimensional arrays in Java are really one-dimensional arrays nested inside a common one-dimensional array raises an interesting issue with respect to how we think about ***compound objects***, which are objects—like a two-dimensional array—that are made up of other objects. In particular, it brings up the question of where a compound object begins and ends.

Thus, if we have a two-dimensional array, a, and another two-dimensional array, b, that has the same entries as a, we probably want to think that a is equal to b. But the one-dimensional arrays that make up the rows of a and b (such as a[0] and b[0]) are stored in different memory locations, even though they have the same internal content. Therefore, a call to the method java.util.Arrays.equals(a,b) will return **false** in this case, because it tests a[k].equals(b[k]), which invokes the Object class's definition of equals.

To support the more natural notion of multidimensional arrays being equal if they have equal contents, the class provides an additional method:

Arrays.deepEquals(a,b):	Identical to Arrays.equals(a,b) except when the elements of a and b are themselves arrays, in which case it calls Arrays.deepEquals(a[k],b[k]) for corresponding entries, rather than a[k].equals(b[k]).

3.5.2 Equivalence Testing with Linked Lists

In this section, we develop an implementation of the equals method in the context of the SinglyLinkedList class of Section 3.2.1. Using a definition very similar to the treatment of arrays by the java.util.Arrays.equals method, we consider two lists to be equivalent if they have the same length and contents that are element-by-element equivalent. We can evaluate such equivalence by simultaneously traversing two lists, verifying that x.equals(y) for each pair of corresponding elements x and y.

The implementation of the SinglyLinkedList.equals method is given in Code Fragment 3.19. Although we are focused on comparing two singly linked lists, the equals method must take an arbitrary Object as a parameter. We take a conservative approach, demanding that two objects be instances of the same class to have any possibility of equivalence. (For example, we do not consider a singly linked list to be equivalent to a doubly linked list with the same sequence of elements.) After ensuring, at line 2, that parameter o is nonnull, line 3 uses the getClass( ) method supported by all objects to test whether the two instances belong to the same class.

When reaching line 4, we have ensured that the parameter was an instance of the SinglyLinkedList class (or an appropriate subclass), and so we can safely cast it to a SinglyLinkedList, so that we may access its instance variables size and head. There is subtlety involving the treatment of Java's generics framework. Although our SinglyLinkedList class has a declared formal type parameter <E>, we cannot detect at runtime whether the other list has a matching type. (For those interested, look online for a discussion of ***erasure*** in Java.) So we revert to using a more classic approach with nonparameterized type SinglyLinkedList at line 4, and nonparameterized Node declarations at lines 6 and 7. If the two lists have incompatible types, this will be detected when calling the equals method on corresponding elements.

```
 1  public boolean equals(Object o) {
 2    if (o == null) return false;
 3    if (getClass( ) != o.getClass( )) return false;
 4    SinglyLinkedList other = (SinglyLinkedList) o;      // use nonparameterized type
 5    if (size != other.size) return false;
 6    Node walkA = head;                                  // traverse the primary list
 7    Node walkB = other.head;                            // traverse the secondary list
 8    while (walkA != null) {
 9      if (!walkA.getElement( ).equals(walkB.getElement( ))) return false; //mismatch
10      walkA = walkA.getNext( );
11      walkB = walkB.getNext( );
12    }
13    return true;     // if we reach this, everything matched successfully
14  }
```

Code Fragment 3.19: Implementation of the SinglyLinkedList.equals method.

3.6 Cloning Data Structures

The beauty of object-oriented programming is that abstraction allows for a data structure to be treated as a single object, even though the encapsulated implementation of the structure might rely on a more complex combination of many objects. In this section, we consider what it means to make a copy of such a structure.

In a programming environment, a common expectation is that a copy of an object has its own state and that, once made, the copy is independent of the original (for example, so that changes to one do not directly affect the other). However, when objects have fields that are reference variables pointing to auxiliary objects, it is not always obvious whether a copy should have a corresponding field that refers to the same auxiliary object, or to a new copy of that auxiliary object.

For example, if a hypothetical AddressBook class has instances that represent an electronic address book—with contact information (such as phone numbers and email addresses) for a person's friends and acquaintances—how might we envision a copy of an address book? Should an entry added to one book appear in the other? If we change a person's phone number in one book, would we expect that change to be synchronized in the other?

There is no one-size-fits-all answer to questions like this. Instead, each class in Java is responsible for defining whether its instances can be copied, and if so, precisely how the copy is constructed. The universal Object superclass defines a method named **clone**, which can be used to produce what is known as a ***shallow copy*** of an object. This uses the standard assignment semantics to assign the value of each field of the new object equal to the corresponding field of the existing object that is being copied. The reason this is known as a shallow copy is because if the field is a reference type, then an initialization of the form duplicate.field = original.field causes the field of the new object to refer to the same underlying instance as the field of the original object.

A shallow copy is not always appropriate for all classes, and therefore, Java intentionally disables use of the clone( ) method by declaring it as **protected**, and by having it throw a CloneNotSupportedException when called. The author of a class must explicitly declare support for cloning by formally declaring that the class implements the Cloneable interface, and by declaring a public version of the clone( ) method. That public method can simply call the protected one to do the field-by-field assignment that results in a shallow copy, if appropriate. However, for many classes, the class may choose to implement a deeper version of cloning, in which some of the referenced objects are themselves cloned.

For most of the data structures in this book, we omit the implementation of a valid clone method (leaving it as an exercise). However, in this section, we consider approaches for cloning both arrays and linked lists, including a concrete implementation of the clone method for the SinglyLinkedList class.

3.6.1 Cloning Arrays

Although arrays support some special syntaxes such as a[k] and a.length, it is important to remember that they are objects, and that array variables are reference variables. This has important consequences. As a first example, consider the following code:

```
int[ ] data = {2, 3, 5, 7, 11, 13, 17, 19};
int[ ] backup;
backup = data;                              // warning; not a copy
```

The assignment of variable backup to data does not create any new array; it simply creates a new alias for the same array, as portrayed in Figure 3.23.

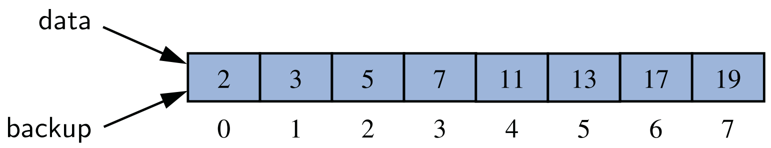


Figure 3.23: The result of the command backup = data for **int** arrays.

Instead, if we want to make a copy of the array, data, and assign a reference to the new array to variable, backup, we should write:

```
backup = data.clone( );
```

The clone method, when executed on an array, initializes each cell of the new array to the value that is stored in the corresponding cell of the original array. This results in an independent array, as shown in Figure 3.24.

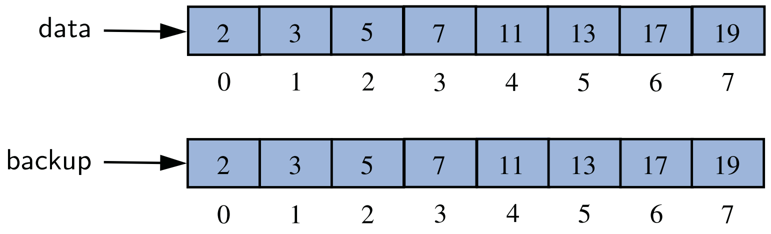


Figure 3.24: The result of the command backup = data.clone( ) for **int** arrays.

If we subsequently make an assignment such as data[4] = 23 in this configuration, the backup array is unaffected.

There are more considerations when copying an array that stores reference types rather than primitive types. The clone( ) method produces a ***shallow copy*** of the array, producing a new array whose cells refer to the same objects referenced by the first array.

For example, if the variable contacts refers to an array of hypothetical Person instances, the result of the command guests = contacts.clone( ) produces a shallow copy, as portrayed in Figure 3.25.

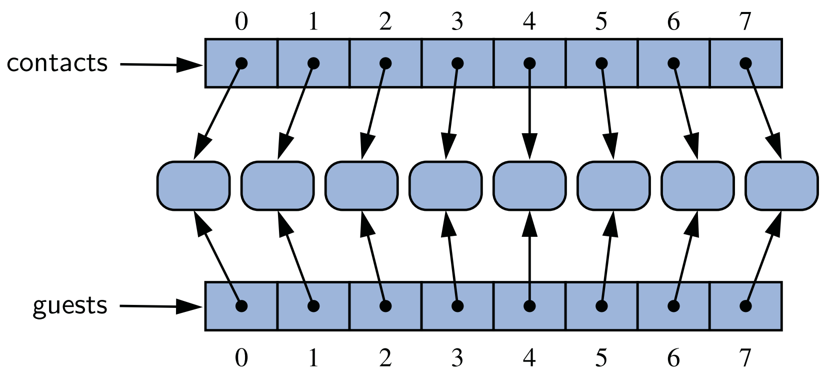


Figure 3.25: A shallow copy of an array of objects, resulting from the command guests = contacts.clone( ).

A ***deep copy*** of the contact list can be created by iteratively cloning the individual elements, as follows, but only if the Person class is declared as Cloneable.

```
Person[ ] guests = new Person[contacts.length];
for (int k=0; k < contacts.length; k++)
  guests[k] = (Person) contacts[k].clone( );        // returns Object type
```

Because a two-dimensional array is really a one-dimensional array storing other one-dimensional arrays, the same distinction between a shallow and deep copy exists. Unfortunately, the java.util.Arrays class does not provide any "deepClone" method. However, we can implement our own method by cloning the individual rows of an array, as shown in Code Fragment 3.20, for a two-dimensional array of integers.

```
public static int[ ][ ] deepClone(int[ ][ ] original) {
  int[ ][ ] backup = new int[original.length][ ];       // create top-level array of arrays
  for (int k=0; k < original.length; k++)
    backup[k] = original[k].clone( );                   // copy row k
  return backup;
}
```

Code Fragment 3.20: A method for creating a deep copy of a two-dimensional array of integers.

3.6.2 Cloning Linked Lists

In this section, we add support for cloning instances of the SinglyLinkedList class from Section 3.2.1. The first step to making a class cloneable in Java is declaring that it implements the Cloneable interface. Therefore, we adjust the first line of the class definition to appear as follows:

```
public class SinglyLinkedList<E> implements Cloneable {
```

The remaining task is implementing a public version of the clone( ) method of the class, which we present in Code Fragment 3.21. By convention, that method should begin by creating a new instance using a call to **super**.clone( ), which in our case invokes the method from the Object class (line 3). Because the inherited version returns an Object, we perform a narrowing cast to type SinglyLinkedList<E>.

At this point in the execution, the other list has been created as a shallow copy of the original. Since our list class has two fields, size and head, the following assignments have been made:

```
other.size = this.size;
other.head = this.head;
```

While the assignment of the size variable is correct, we cannot allow the new list to share the same head value (unless it is **null**). For a nonempty list to have an independent state, it must have an entirely new chain of nodes, each storing a reference to the corresponding element from the original list. We therefore create a new head node at line 5 of the code, and then perform a walk through the remainder of the original list (lines 8–13) while creating and linking new nodes for the new list.

```
1   public SinglyLinkedList<E> clone() throws CloneNotSupportedException {
2     // always use inherited Object.clone() to create the initial copy
3     SinglyLinkedList<E> other = (SinglyLinkedList<E>) super.clone(); // safe cast
4     if (size > 0) {                          // we need independent chain of nodes
5       other.head = new Node<>(head.getElement(), null);
6       Node<E> walk = head.getNext();         // walk through remainder of original list
7       Node<E> otherTail = other.head;        // remember most recently created node
8       while (walk != null) {                 // make a new node storing same element
9         Node<E> newest = new Node<>(walk.getElement(), null);
10        otherTail.setNext(newest);           // link previous node to this one
11        otherTail = newest;
12        walk = walk.getNext();
13      }
14    }
15    return other;
16  }
```

Code Fragment 3.21: Implementation of the SinglyLinkedList.clone method.

3.7 Exercises

Reinforcement

R-3.1 Give the next five pseudorandom numbers generated by the process described on page 113, with a = 12, b = 5, and n = 100, and 92 as the seed for cur.

R-3.2 Write a Java method that repeatedly selects and removes a random entry from an array until the array holds no more entries.

R-3.3 Explain the changes that would have to be made to the program of Code Fragment 3.8 so that it could perform the Caesar cipher for messages that are written in an alphabet-based language other than English, such as Greek, Russian, or Hebrew.

R-3.4 The TicTacToe class of Code Fragments 3.9 and 3.10 has a flaw, in that it allows a player to place a mark even after the game has already been won by someone. Modify the class so that the putMark method throws an IllegalStateException in that case.

R-3.5 The removeFirst method of the SinglyLinkedList class includes a special case to reset the tail field to **null** when deleting the last node of a list (see lines 51 and 52 of Code Fragment 3.15). What are the consequences if we were to remove those two lines from the code? Explain why the class would or would not work with such a modification.

R-3.6 Give an algorithm for finding the second-to-last node in a singly linked list in which the last node is indicated by a null next reference.

R-3.7 Consider the implementation of CircularlyLinkedList.addFirst, in Code Fragment 3.16. The else body at lines 39 and 40 of that method relies on a locally declared variable, newest. Redesign that clause to avoid use of any local variable.

R-3.8 Describe a method for finding the middle node of a doubly linked list with header and trailer sentinels by "link hopping," and without relying on explicit knowledge of the size of the list. In the case of an even number of nodes, report the node slightly left of center as the "middle." What is the running time of this method?

R-3.9 Give an implementation of the size( ) method for the SingularlyLinkedList class, assuming that we did not maintain size as an instance variable.

R-3.10 Give an implementation of the size( ) method for the CircularlyLinkedList class, assuming that we did not maintain size as an instance variable.

R-3.11 Give an implementation of the size( ) method for the DoublyLinkedList class, assuming that we did not maintain size as an instance variable.

R-3.12 Implement a rotate( ) method in the SinglyLinkedList class, which has semantics equal to addLast(removeFirst( )), yet without creating any new node.

R-3.13 What is the difference between a shallow equality test and a deep equality test between two Java arrays, A and B, if they are one-dimensional arrays of type **int**? What if the arrays are two-dimensional arrays of type **int**?

R-3.14 Give three different examples of a single Java statement that assigns variable, backup, to a new array with copies of all **int** entries of an existing array, original.

R-3.15 Implement the equals( ) method for the CircularlyLinkedList class, assuming that two lists are equal if they have the same sequence of elements, with corresponding elements currently at the front of the list.

R-3.16 Implement the equals( ) method for the DoublyLinkedList class.

Creativity

C-3.17 Let A be an array of size $n \geq 2$ containing integers from 1 to $n-1$ inclusive, one of which is repeated. Describe an algorithm for finding the integer in A that is repeated.

C-3.18 Let B be an array of size $n \geq 6$ containing integers from 1 to $n-5$ inclusive, five of which are repeated. Describe an algorithm for finding the five integers in B that are repeated.

C-3.19 Give Java code for performing add(e) and remove(i) methods for the Scoreboard class, as in Code Fragments 3.3 and 3.4, except this time, don't maintain the game entries in order. Assume that we still need to keep n entries stored in indices 0 to $n-1$. You should be able to implement the methods without using any loops, so that the number of steps they perform does not depend on n.

C-3.20 Give examples of values for a and b in the pseudorandom generator given on page 113 of this chapter such that the result is not very random looking, for n = 1000.

C-3.21 Suppose you are given an array, A, containing 100 integers that were generated using the method r.nextInt(10), where r is an object of type java.util.Random. Let x denote the product of the integers in A. There is a single number that x will equal with probability at least 0.99. What is that number and what is a formula describing the probability that x is equal to that number?

C-3.22 Write a method, shuffle(A), that rearranges the elements of array A so that every possible ordering is equally likely. You may rely on the nextInt(n) method of the java.util.Random class, which returns a random number between 0 and $n-1$ inclusive.

C-3.23 Suppose you are designing a multiplayer game that has $n \geq 1000$ players, numbered 1 to n, interacting in an enchanted forest. The winner of this game is the first player who can meet all the other players at least once (ties are allowed). Assuming that there is a method meet(i, j), which is called each time a player i meets a player j (with $i \neq j$), describe a way to keep track of the pairs of meeting players and who is the winner.

C-3.24 Write a Java method that takes two three-dimensional integer arrays and adds them componentwise.

C-3.25 Describe an algorithm for concatenating two singly linked lists L and M, into a single list L' that contains all the nodes of L followed by all the nodes of M.

C-3.26 Give an algorithm for concatenating two doubly linked lists L and M, with header and trailer sentinel nodes, into a single list L'.

C-3.27 Describe in detail how to swap two nodes x and y (and not just their contents) in a singly linked list L given references only to x and y. Repeat this exercise for the case when L is a doubly linked list. Which algorithm takes more time?

C-3.28 Describe in detail an algorithm for reversing a singly linked list L using only a constant amount of additional space.

C-3.29 Suppose you are given two circularly linked lists, L and M. Describe an algorithm for telling if L and M store the same sequence of elements (but perhaps with different starting points).

C-3.30 Given a circularly linked list L containing an even number of nodes, describe how to split L into two circularly linked lists of half the size.

C-3.31 Our implementation of a doubly linked list relies on two sentinel nodes, header and trailer, but a single sentinel node that guards both ends of the list should suffice. Reimplement the DoublyLinkedList class using only one sentinel node.

C-3.32 Implement a circular version of a doubly linked list, without any sentinels, that supports all the public behaviors of the original as well as two new update methods, rotate( ) and rotateBackward( ).

C-3.33 Solve the previous problem using inheritance, such that a DoublyLinkedList class inherits from the existing CircularlyLinkedList, and the DoublyLinkedList.Node nested class inherits from CircularlyLinkedList.Node.

C-3.34 Implement the clone( ) method for the CircularlyLinkedList class.

C-3.35 Implement the clone( ) method for the DoublyLinkedList class.

Projects

P-3.36 Write a Java program for a matrix class that can add and multiply arbitrary two-dimensional arrays of integers.

P-3.37 Write a class that maintains the top ten scores for a game application, implementing the add and remove methods of Section 3.1.1, but using a singly linked list instead of an array.

P-3.38 Perform the previous project, but use a doubly linked list. Moreover, your implementation of remove(i) should make the fewest number of pointer hops to get to the game entry at index i.

P-3.39 Write a program that can perform the Caesar cipher for English messages that include both upper- and lowercase characters.

P-3.40 Implement a class, SubstitutionCipher, with a constructor that takes a string with the 26 uppercase letters in an arbitrary order and uses that as the encoder for a cipher (that is, A is mapped to the first character of the parameter, B is mapped to the second, and so on.) You should derive the decoding map from the forward version.

P-3.41 Redesign the CaesarCipher class as a subclass of the SubstitutionCipher from the previous problem.

P-3.42 Design a RandomCipher class as a subclass of the SubstitutionCipher from Exercise P-3.40, so that each instance of the class relies on a random permutation of letters for its mapping.

P-3.43 In the children's game, Duck, Duck, Goose, a group of children sit in a circle. One of them is elected "it" and that person walks around the outside of the circle. The person who is "it" pats each child on the head, saying "Duck" each time, until randomly reaching a child that the "it" person identifies as "Goose." At this point there is a mad scramble, as the "Goose" and the "it" person race around the circle. Whoever returns to the Goose's former place first gets to remain in the circle. The loser of this race is the "it" person for the next round of play. The game continues like this until the children get bored or an adult tells them it's snack time. Write software that simulates a game of Duck, Duck, Goose.

Chapter Notes

The fundamental data structures of arrays and linked lists discussed in this chapter belong to the folklore of computer science. They were first chronicled in the computer science literature by Knuth in his seminal book on *Fundamental Algorithms* [60].