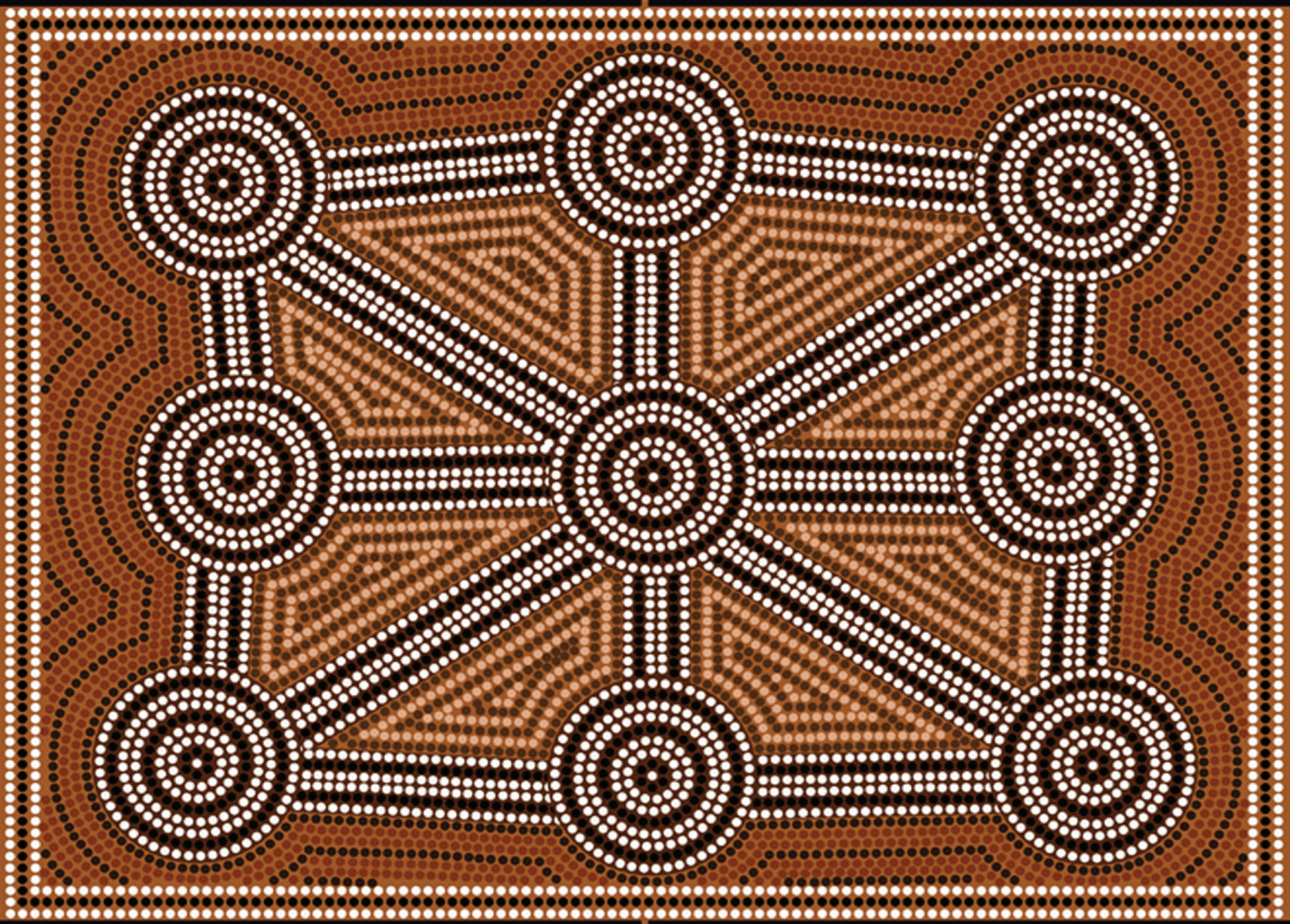
SIXTH EDITION

DATA STRUCTURES & ALGORITHMS



MICHAEL T. GOODRICH
ROBERTO TAMASSIA
MICHAEL H. GOLDWASSER

in JAVA™

WILEY

Data Structures and Algorithms in Java™

Sixth Edition

Michael T. Goodrich
Department of Computer Science
University of California, Irvine

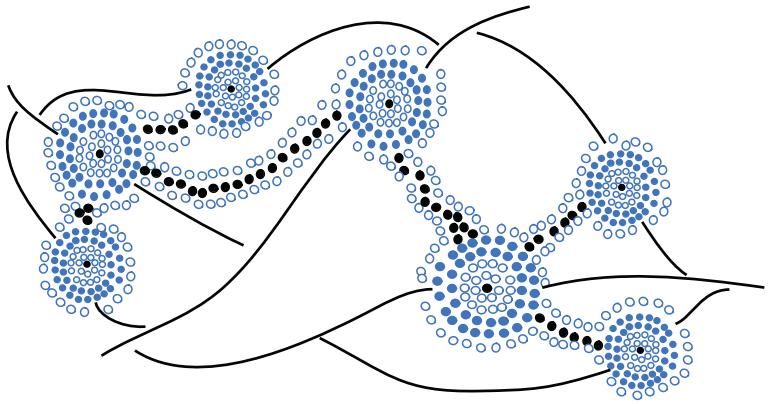
Roberto Tamassia
Department of Computer Science
Brown University

Michael H. Goldwasser
Department of Mathematics and Computer Science
Saint Louis University

WILEY

Contents

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.

```
1  public class GameEntry {
2    private String name;                    // name of the person earning this score
3    private int score;                      // the score value
4    /** Constructs a game entry with given parameters.. */
5    public GameEntry(String n, int s) {
6      name = n;
7      score = s;
8    }
9    /** Returns the name field. */
10   public String getName( ) { return name; }
11   /** Returns the score field. */
12   public int getScore( ) { return score; }
13   /** Returns a string representation of this entry. */
14   public String toString( ) {
15     return "(" + name + ", " + score + ")";
16   }
17 }
```

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.

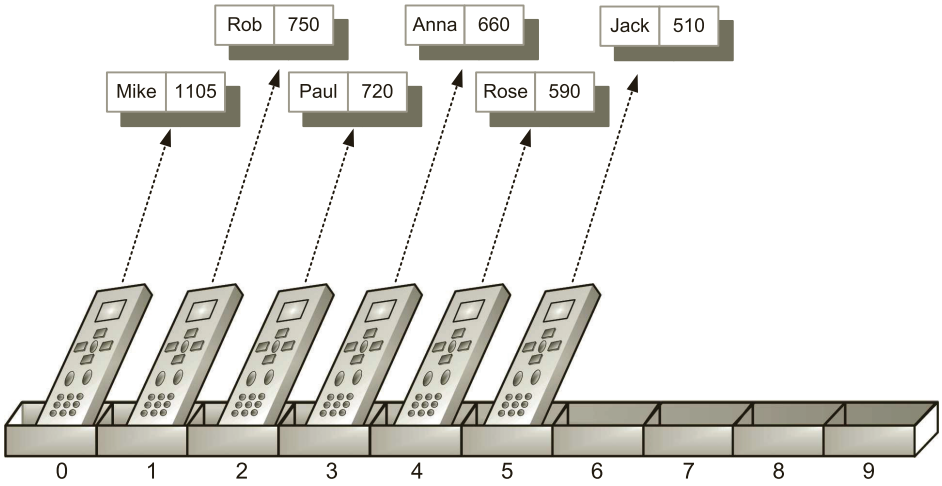


Figure 3.1: An illustration of an array of length ten storing references to six GameEntry objects in the cells with indices 0 to 5; the rest are **null** references.

```
1   /** Class for storing high scores in an array in nondecreasing order. */
2   public class Scoreboard {
3     private int numEntries = 0;                // number of actual entries
4     private GameEntry[ ] board;                // array of game entries (names & scores)
5     /** Constructs an empty scoreboard with the given capacity for storing entries. */
6     public Scoreboard(int capacity) {
7       board = new GameEntry[capacity];
8     }
...   // more methods will go here
36  }
```

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.

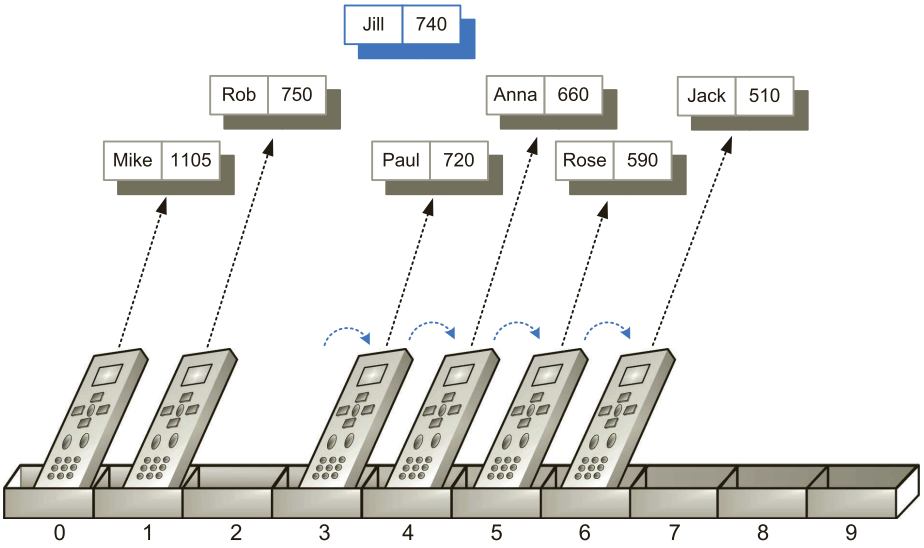


Figure 3.2: Preparing to add Jill's GameEntry object to the board array. In order to make room for the new reference, we have to shift any references to game entries with smaller scores than the new one to the right by one cell.

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.

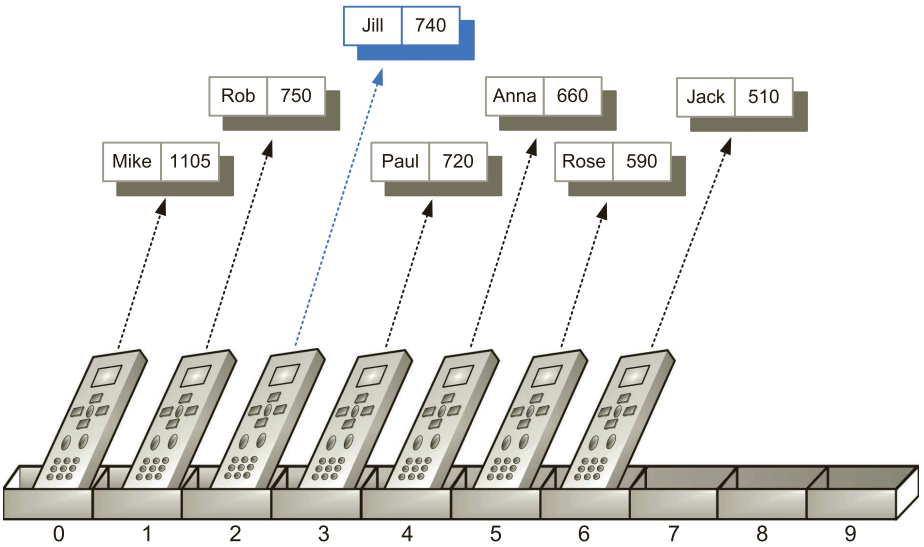


Figure 3.3: Adding a reference to Jill's GameEntry object to the board array. The reference can now be inserted at index 2, since we have shifted all references to GameEntry objects with scores less than the new one to the right.

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.)

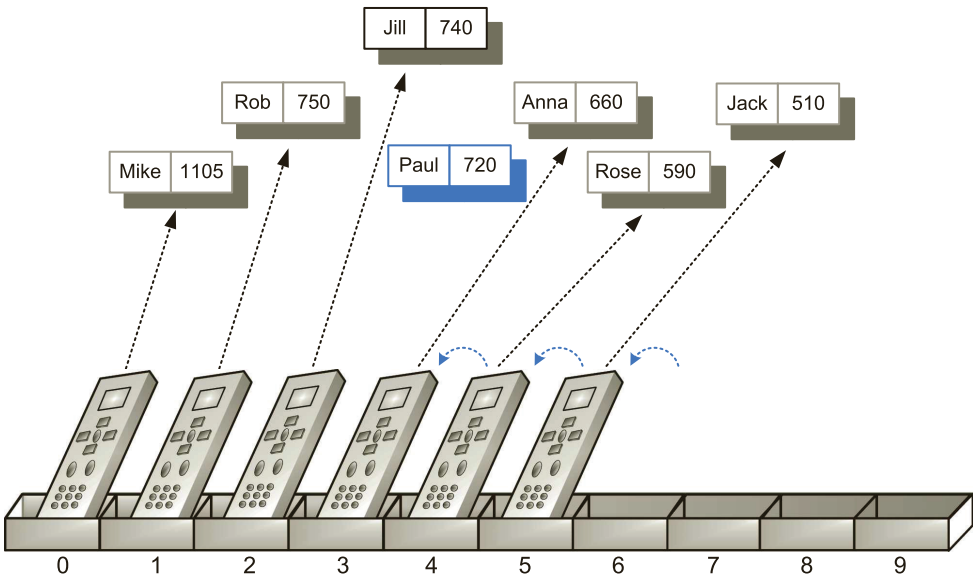


Figure 3.4: An illustration of the removal of Paul's score from index 3 of an array storing references to GameEntry objects.

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.

```
25   /** Remove and return the high score at index i. */
26   public GameEntry remove(int i) throws IndexOutOfBoundsException {
27     if (i < 0 || i >= numEntries)
28       throw new IndexOutOfBoundsException("Invalid index: " + i);
29     GameEntry temp = board[i];                  // save the object to be removed
30     for (int j = i; j < numEntries - 1; j++)    // count up from i (not down)
31       board[j] = board[j+1];                    // move one cell to the left
32     board[numEntries -1 ] = null;               // null out the old last score
33     numEntries--;
34     return temp;                                // return the removed object
35   }
```

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.

```
1   /** Insertion-sort of an array of characters into nondecreasing order */
2   public static void insertionSort(char[ ] data) {
3     int n = data.length;
4     for (int k = 1; k < n; k++) {                  // begin with second character
5       char cur = data[k];                          // time to insert cur=data[k]
6       int j = k;                                   // find correct index j for cur
7       while (j > 0 && data[j−1] > cur) {           // thus, data[j-1] must go after cur
8         data[j] = data[j−1];                       // slide data[j-1] rightward
9         j−−;                                       // and consider previous j for cur
10      }
11      data[j] = cur;                               // this is the proper place for cur
12    }
13  }
```

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

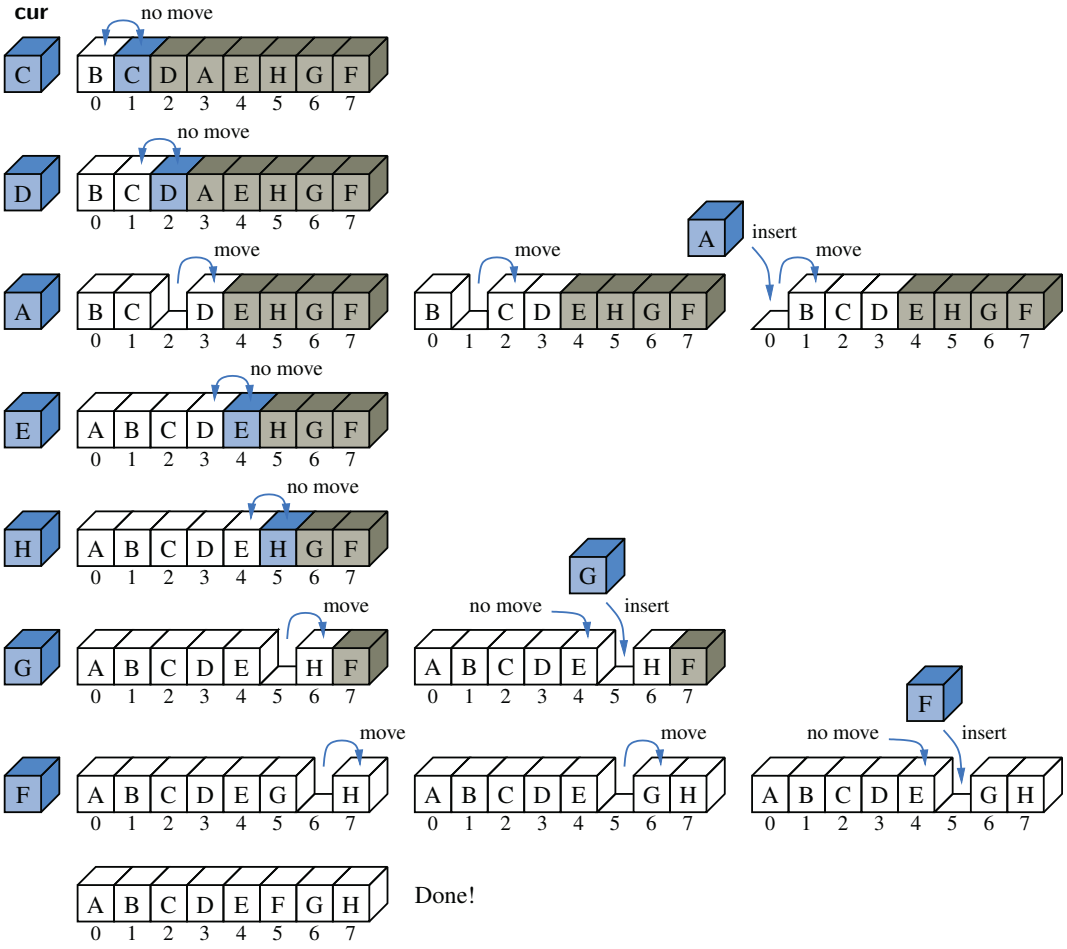


Figure 3.5: Execution of the insertion-sort algorithm on an array of eight characters. Each row corresponds to an iteration of the outer loop, and each copy of the sequence in a row corresponds to an iteration of the inner loop. The current element that is being inserted is highlighted in the array,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

$$\mathsf{next} = (\mathsf{a} * \mathsf{cur} + \mathsf{b}) \ \% \ \mathsf{n};$$

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 $\mathsf{n} = 2^{48}$. 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.

```
1  import java.util.Arrays;
2  import java.util.Random;
3  /** Program showing some array uses. */
4  public class ArrayTest {
5    public static void main(String[ ] args) {
6      int data[ ] = new int[10];
7      Random rand = new Random( );            // a pseudo-random number generator
8      rand.setSeed(System.currentTimeMillis( ));          // use current time as a seed
9      // fill the data array with pseudo-random numbers from 0 to 99, inclusive
10     for (int i = 0; i < data.length; i++)
11       data[i] = rand.nextInt(100);                  // the next pseudo-random number
12     int[ ] orig = Arrays.copyOf(data, data.length); // make a copy of the data array
13     System.out.println("arrays equal before sort: "+Arrays.equals(data, orig));
14     Arrays.sort(data);                    // sorting the data array (orig is unchanged)
15     System.out.println("arrays equal after sort: " + Arrays.equals(data, orig));
16     System.out.println("orig = " + Arrays.toString(orig));
17     System.out.println("data = " + Arrays.toString(data));
18   }
19 }
```

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.

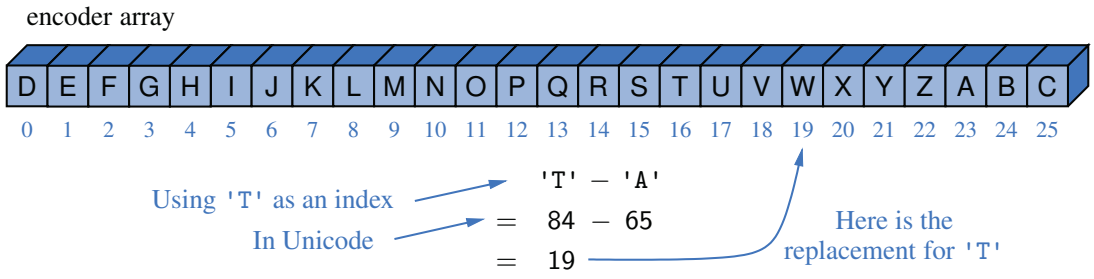


Figure 3.6: Illustrating the use of uppercase characters as indices, in this case to perform the replacement rule for Caesar cipher encryption.

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 $(k + r)$ mod 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 $(k - r + 26)$ mod 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.
```

```
1  /** Class for doing encryption and decryption using the Caesar Cipher. */
2  public class CaesarCipher {
3    protected char[ ] encoder = new char[26];          // Encryption array
4    protected char[ ] decoder = new char[26];          // Decryption array
5    /** Constructor that initializes the encryption and decryption arrays */
6    public CaesarCipher(int rotation) {
7      for (int k=0; k < 26; k++) {
8        encoder[k] = (char) ('A' + (k + rotation) % 26);
9        decoder[k] = (char) ('A' + (k - rotation + 26) % 26);
10     }
11   }
12   /** Returns String representing encrypted message. */
13   public String encrypt(String message) {
14     return transform(message, encoder);               // use encoder array
15   }
16   /** Returns decrypted message given encrypted secret. */
17   public String decrypt(String secret) {
18     return transform(secret, decoder);                // use decoder array
19   }
20   /** Returns transformation of original String using given code. */
21   private String transform(String original, char[ ] code) {
22     char[ ] msg = original.toCharArray( );
23     for (int k=0; k < msg.length; k++)
24       if (Character.isUpperCase(msg[k])) {            // we have a letter to change
25         int j = msg[k] - 'A';                         // will be value from 0 to 25
26         msg[k] = code[j];                             // replace the character
27       }
28     return new String(msg);
29   }
30   /** Simple main method for testing the Caesar cipher */
31   public static void main(String[ ] args) {
32     CaesarCipher cipher = new CaesarCipher(3);
33     System.out.println("Encryption code = " + new String(cipher.encoder));
34     System.out.println("Decryption code = " + new String(cipher.decoder));
35     String message = "THE EAGLE IS IN PLAY; MEET AT JOE'S.";
36     String coded = cipher.encrypt(message);
37     System.out.println("Secret:  " + coded);
38     String answer = cipher.decrypt(coded);
39     System.out.println("Message: " + answer);        // should be plaintext again
40   }
41 }
```

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.

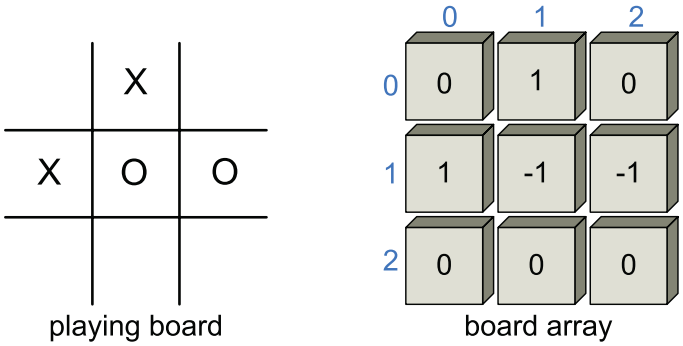


Figure 3.8: An illustration of a Tic-Tac-Toe board and the two-dimensional integer array, board, representing it.

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).

```
1  /** Simulation of a Tic-Tac-Toe game (does not do strategy). */
2  public class TicTacToe {
3    public static final int X = 1, O = -1;        // players
4    public static final int EMPTY = 0;            // empty cell
5    private int board[ ][ ] = new int[3][3];      // game board
6    private int player;                           // current player
7    /** Constructor */
8    public TicTacToe( ) { clearBoard( ); }
9    /** Clears the board */
10   public void clearBoard( ) {
11     for (int i = 0; i < 3; i++)
12       for (int j = 0; j < 3; j++)
13         board[i][j] = EMPTY;                    // every cell should be empty
14     player = X;                                 // the first player is 'X'
15   }
16   /** Puts an X or O mark at position i,j. */
17   public void putMark(int i, int j) throws IllegalArgumentException {
18     if ((i < 0) || (i > 2) || (j < 0) || (j > 2))
19       throw new IllegalArgumentException("Invalid board position");
20     if (board[i][j] != EMPTY)
21       throw new IllegalArgumentException("Board position occupied");
22     board[i][j] = player;              // place the mark for the current player
23     player = - player;                 // switch players (uses fact that O = - X)
24   }
25   /** Checks whether the board configuration is a win for the given player. */
26   public boolean isWin(int mark) {
27     return ((board[0][0] + board[0][1] + board[0][2] == mark*3)      // row 0
28             || (board[1][0] + board[1][1] + board[1][2] == mark*3)   // row 1
29             || (board[2][0] + board[2][1] + board[2][2] == mark*3)   // row 2
30             || (board[0][0] + board[1][0] + board[2][0] == mark*3)   // column 0
31             || (board[0][1] + board[1][1] + board[2][1] == mark*3)   // column 1
32             || (board[0][2] + board[1][2] + board[2][2] == mark*3)   // column 2
33             || (board[0][0] + board[1][1] + board[2][2] == mark*3)   // diagonal
34             || (board[2][0] + board[1][1] + board[0][2] == mark*3)); // rev diag
35   }
36   /** Returns the winning player's code, or 0 to indicate a tie (or unfinished game).*/
37   public int winner( ) {
38     if (isWin(X))
39       return(X);
40     else if (isWin(O))
41       return(O);
42     else
43       return(0);
44   }
```

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

```
45   /** Returns a simple character string showing the current board. */
46   public String toString( ) {
47     StringBuilder sb = new StringBuilder( );
48     for (int i=0; i<3; i++) {
49       for (int j=0; j<3; j++) {
50         switch (board[i][j]) {
51         case X:          sb.append("X"); break;
52         case O:          sb.append("O"); break;
53         case EMPTY:    sb.append("  "); break;
54         }
55         if (j < 2) sb.append("|");                  // column boundary
56       }
57       if (i < 2) sb.append("\n-----\n");            // row boundary
58     }
59     return sb.toString( );
60   }
61   /** Test run of a simple game */
62   public static void main(String[ ] args) {
63     TicTacToe game = new TicTacToe( );
64     /* X moves: */                 /* O moves: */
65     game.putMark(1,1);             game.putMark(0,2);
66     game.putMark(2,2);             game.putMark(0,0);
67     game.putMark(0,1);             game.putMark(2,1);
68     game.putMark(1,2);             game.putMark(1,0);
69     game.putMark(2,0);
70     System.out.println(game);
71     int winningPlayer = game.winner( );
72     String[ ] outcome = {"O wins", "Tie", "X wins"};  // rely on ordering
73     System.out.println(outcome[1 + winningPlayer]);
74   }
75 }
```

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).

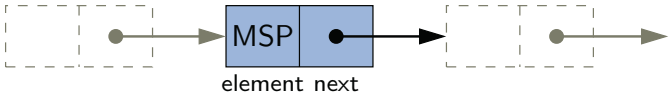


Figure 3.10: Example of a node instance that forms part of a singly linked list. The node's element field refers to an object that is an element of the sequence (the airport code MSP, in this example), while the next field refers to the subsequent node of the linked list (or null if there is no further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

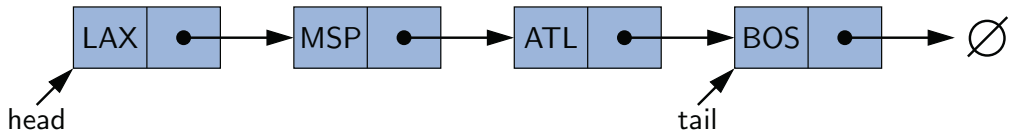


Figure 3.11: Example of a singly linked list whose elements are strings indicating airport codes. The list instance maintains a member named head that refers to the first node of the list, and another member named tail that refers to the last node of the list. The **null** value is denoted as Ø.

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.

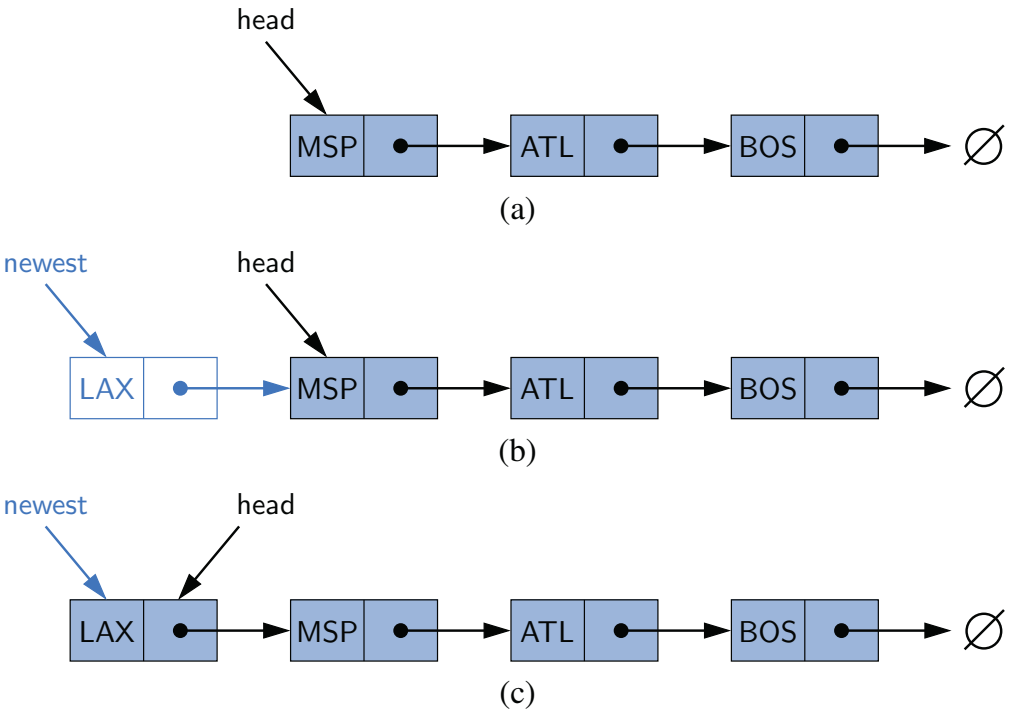


Figure 3.12: Insertion of an element at the head of a singly linked list: (a) before the insertion; (b) after a new node is created and linked to the existing head; (c) after reassignment of the head reference to the newest node.

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.

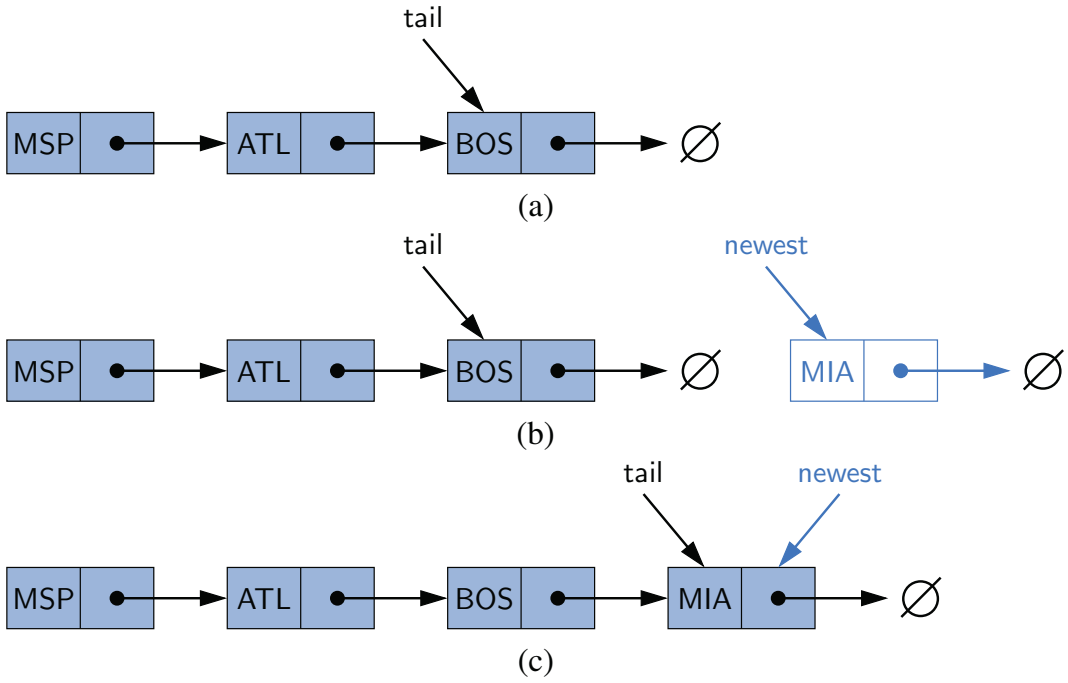


Figure 3.13: Insertion at the tail of a singly linked list: (a) before the insertion; (b) after creation of a new node; (c) after reassignment of the tail reference. Note that we must set the next link of the tail node in (b) before we assign the tail variable to point to the new node in (c).

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.

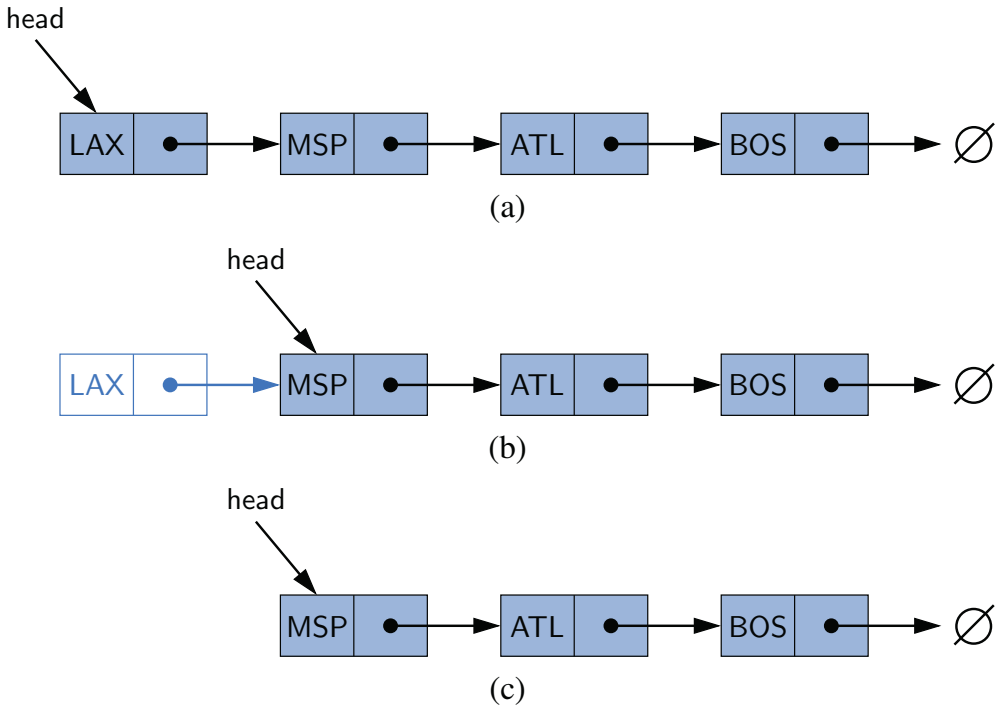


Figure 3.14: Removal of an element at the head of a singly linked list: (a) before the removal; (b) after "linking out" the old head; (c) final configuration.

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.

```
1  public class SinglyLinkedList<E> {
2    //---------------- nested Node class ----------------
3    private static class Node<E> {
4      private E element;                  // reference to the element stored at this node
5      private Node<E> next;               // reference to the subsequent node in the list
6      public Node(E e, Node<E> n) {
7        element = e;
8        next = n;
9      }
10     public E getElement( ) { return element; }
11     public Node<E> getNext( ) { return next; }
12     public void setNext(Node<E> n) { next = n; }
13   } //---------- 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.)

```
1  public class SinglyLinkedList<E> {
...    (nested Node class goes here)
14   // instance variables of the SinglyLinkedList
15   private Node<E> head = null;          // head node of the list (or null if empty)
16   private Node<E> tail = null;          // last node of the list (or null if empty)
17   private int size = 0;                 // number of nodes in the list
18   public SinglyLinkedList( ) { }        // constructs an initially empty list
19   // access methods
20   public int size( ) { return size; }
21   public boolean isEmpty( ) { return size == 0; }
22   public E first( ) {                   // returns (but does not remove) the first element
23     if (isEmpty( )) return null;
24     return head.getElement( );
25   }
26   public E last( ) {                    // returns (but does not remove) the last element
27     if (isEmpty( )) return null;
28     return tail.getElement( );
29   }
30   // update methods
31   public void addFirst(E e) {           // adds element e to the front of the list
32     head = new Node<>(e, head);         // create and link a new node
33     if (size == 0)
34       tail = head;                      // special case: new node becomes tail also
35     size++;
36   }
37   public void addLast(E e) {            // adds element e to the end of the list
38     Node<E> newest = new Node<>(e, null);    // node will eventually be the tail
39     if (isEmpty( ))
40       head = newest;                    // special case: previously empty list
41     else
42       tail.setNext(newest);             // new node after existing tail
43     tail = newest;                      // new node becomes the tail
44     size++;
45   }
46   public E removeFirst( ) {             // removes and returns the first element
47     if (isEmpty( )) return null;        // nothing to remove
48     E answer = head.getElement( );
49     head = head.getNext( );             // will become null if list had only one node
50     size--;
51     if (size == 0)
52       tail = null;                      // special case as list is now empty
53     return answer;
54   }
55 }
```

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.

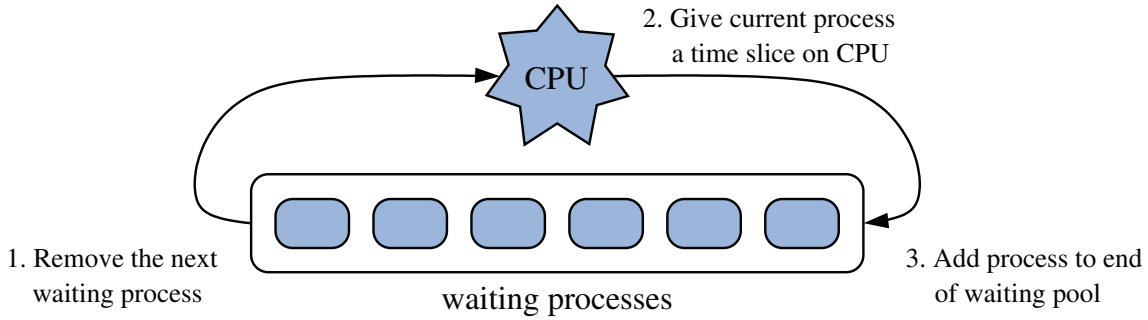


Figure 3.15: The three iterative steps for round-robin scheduling.

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.

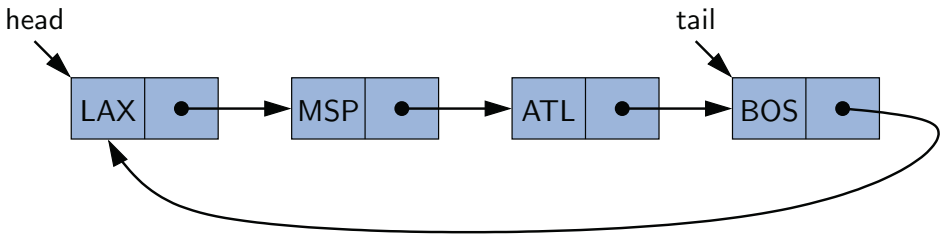


Figure 3.16: Example of a singly linked list with circular structure.

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.

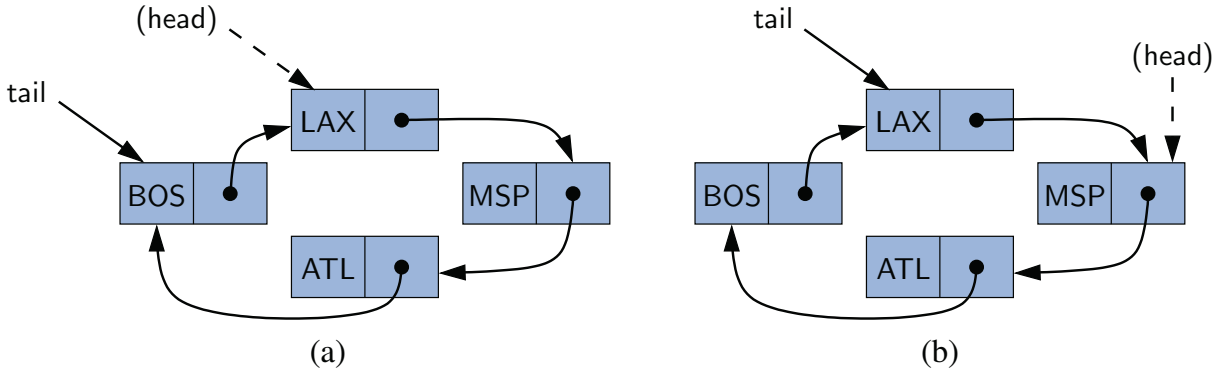


Figure 3.17: The rotation operation on a circularly linked list: (a) before the rotation, representing sequence { LAX, MSP, ATL, BOS }; (b) after the rotation, representing sequence { MSP, ATL, BOS, LAX }. We display the implicit head reference, which is identified only as tail.getNext( ) within the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

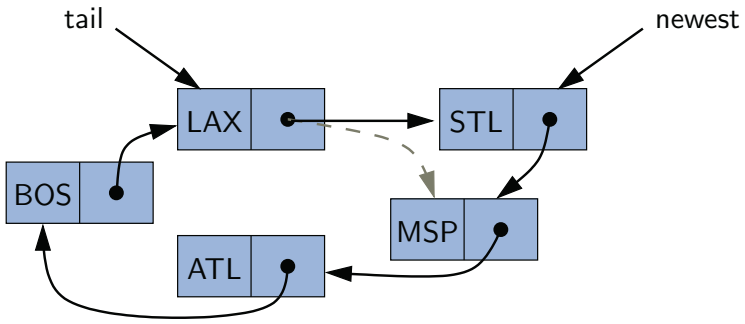


Figure 3.18: Effect of a call to addFirst(STL) on the circularly linked list of Figure 3.17(b). The variable newest has local scope during the execution of the method. Notice that when the operation is complete, STL is the first element of the list, as it is stored within the implicit head, tail.getNext(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

```
1  public class CircularlyLinkedList<E> {
...    (nested node class identical to that of the SinglyLinkedList class)
14   // instance variables of the CircularlyLinkedList
15   private Node<E> tail = null;                // we store tail (but not head)
16   private int size = 0;                       // number of nodes in the list
17   public CircularlyLinkedList( ) { }          // constructs an initially empty list
18   // access methods
19   public int size( ) { return size; }
20   public boolean isEmpty( ) { return size == 0; }
21   public E first( ) {                 // returns (but does not remove) the first element
22     if (isEmpty( )) return null;
23     return tail.getNext( ).getElement( );     // the head is *after* the tail
24   }
25   public E last( ) {                  // returns (but does not remove) the last element
26     if (isEmpty( )) return null;
27     return tail.getElement( );
28   }
29   // update methods
30   public void rotate( ) {             // rotate the first element to the back of the list
31     if (tail != null)                         // if empty, do nothing
32       tail = tail.getNext( );                 // the old head becomes the new tail
33   }
34   public void addFirst(E e) {                 // adds element e to the front of the list
35     if (size == 0) {
36       tail = new Node<>(e, null);
37       tail.setNext(tail);                     // link to itself circularly
38     } else {
39       Node<E> newest = new Node<>(e, tail.getNext( ));
40       tail.setNext(newest);
41     }
42     size++;
43   }
44   public void addLast(E e) {                  // adds element e to the end of the list
45     addFirst(e);                              // insert new element at front of list
46     tail = tail.getNext( );                   // now new element becomes the tail
47   }
48   public E removeFirst( ) {                   // removes and returns the first element
49     if (isEmpty( )) return null;              // nothing to remove
50     Node<E> head = tail.getNext( );
51     if (head == tail) tail = null;            // must be the only node left
52     else tail.setNext(head.getNext( ));       // removes "head" from the list
53     size--;
54     return head.getElement( );
55   }
56 }
```

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.

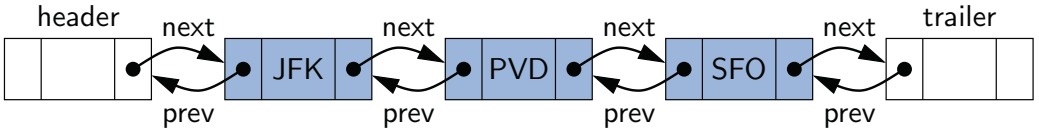


Figure 3.19: A doubly linked list representing the sequence { JFK, PVD, SFO }, using sentinels header and trailer to demarcate the ends of the list.

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.)

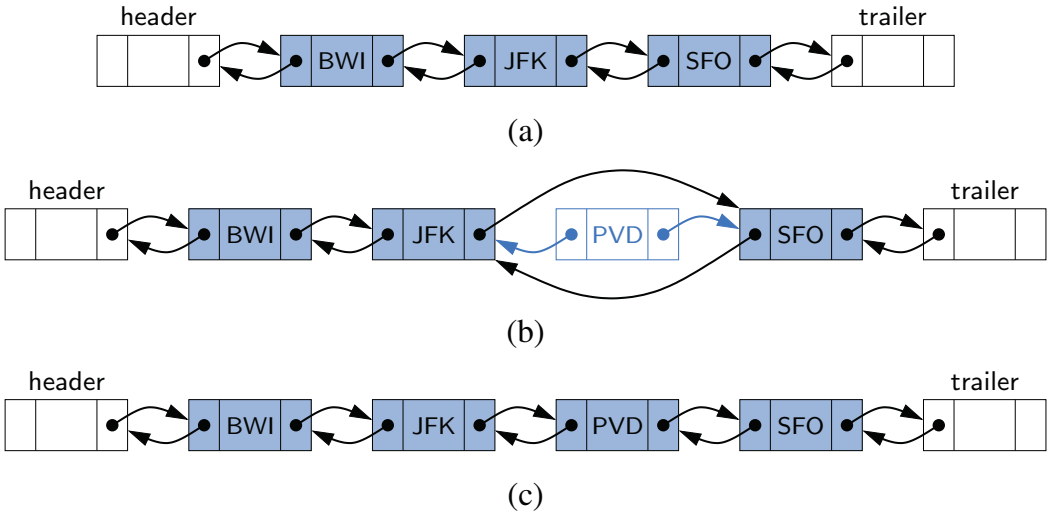


Figure 3.20: Adding an element to a doubly linked list with header and trailer sentinels: (a) before the operation; (b) after creating the new node; (c) after linking the neighbors to the new node.

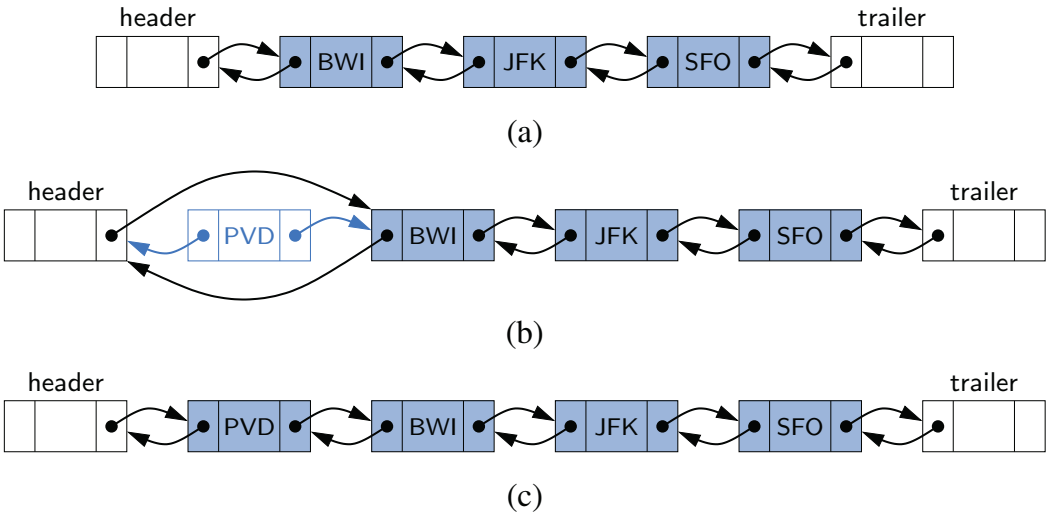


Figure 3.21: Adding an element to the front of a sequence represented by a doubly linked list with header and trailer sentinels: (a) before the operation; (b) after creating the new node; (c) after linking the neighbors to the new node.

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.

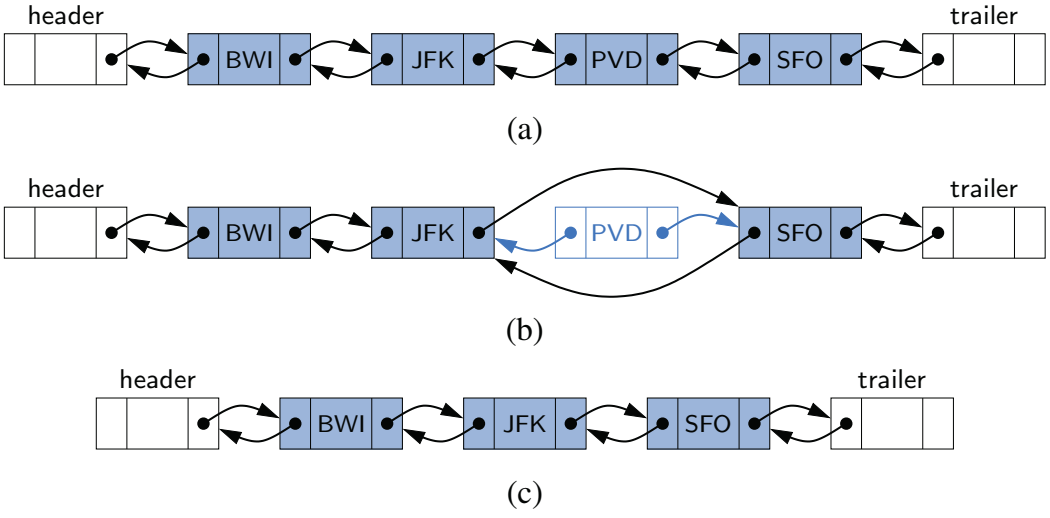


Figure 3.22: Removing the element PVD from a doubly linked list: (a) before the removal; (b) after linking out the old node; (c) after the removal (and garbage collection).

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.

```
1  /** A basic doubly linked list implementation. */
2  public class DoublyLinkedList<E> {
3    //---------------- nested Node class ----------------
4    private static class Node<E> {
5      private E element;                  // reference to the element stored at this node
6      private Node<E> prev;               // reference to the previous node in the list
7      private Node<E> next;               // reference to the subsequent node in the list
8      public Node(E e, Node<E> p, Node<E> n) {
9        element = e;
10       prev = p;
11       next = n;
12     }
13     public E getElement( ) { return element; }
14     public Node<E> getPrev( ) { return prev; }
15     public Node<E> getNext( ) { return next; }
16     public void setPrev(Node<E> p) { prev = p; }
17     public void setNext(Node<E> n) { next = n; }
18   } //----------- end of nested Node class -----------
19
20   // instance variables of the DoublyLinkedList
21   private Node<E> header;                          // header sentinel
22   private Node<E> trailer;                         // trailer sentinel
23   private int size = 0;                            // number of elements in the list
24   /** Constructs a new empty list. */
25   public DoublyLinkedList( ) {
26     header = new Node<>(null, null, null);         // create header
27     trailer = new Node<>(null, header, null);      // trailer is preceded by header
28     header.setNext(trailer);                       // header is followed by trailer
29   }
30   /** Returns the number of elements in the linked list. */
31   public int size( ) { return size; }
32   /** Tests whether the linked list is empty. */
33   public boolean isEmpty( ) { return size == 0; }
34   /** Returns (but does not remove) the first element of the list. */
35   public E first( ) {
36     if (isEmpty( )) return null;
37     return header.getNext( ).getElement( );        // first element is beyond header
38   }
39   /** Returns (but does not remove) the last element of the list. */
40   public E last( ) {
41     if (isEmpty( )) return null;
42     return trailer.getPrev( ).getElement( );       // last element is before trailer
43   }
```

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

```
44    // public update methods
45    /** Adds element e to the front of the list. */
46    public void addFirst(E e) {
47      addBetween(e, header, header.getNext( ));          // place just after the header
48    }
49    /** Adds element e to the end of the list. */
50    public void addLast(E e) {
51      addBetween(e, trailer.getPrev( ), trailer);        // place just before the trailer
52    }
53    /** Removes and returns the first element of the list. */
54    public E removeFirst( ) {
55      if (isEmpty( )) return null;                       // nothing to remove
56      return remove(header.getNext( ));                  // first element is beyond header
57    }
58    /** Removes and returns the last element of the list. */
59    public E removeLast( ) {
60      if (isEmpty( )) return null;                       // nothing to remove
61      return remove(trailer.getPrev( ));                 // last element is before trailer
62    }
63
64    // private update methods
65    /** Adds element e to the linked list in between the given nodes. */
66    private void addBetween(E e, Node<E> predecessor, Node<E> successor) {
67      // create and link a new node
68      Node<E> newest = new Node<>(e, predecessor, successor);
69      predecessor.setNext(newest);
70      successor.setPrev(newest);
71      size++;
72    }
73    /** Removes the given node from the list and returns its element. */
74    private E remove(Node<E> node) {
75      Node<E> predecessor = node.getPrev( );
76      Node<E> successor = node.getNext( );
77      predecessor.setNext(successor);
78      successor.setPrev(predecessor);
79      size--;
80      return node.getElement( );
81    }
82  } //---------- 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.)

5.2 Analyzing Recursive Algorithms

In Chapter 4, we introduced mathematical techniques for analyzing the efficiency of an algorithm, based upon an estimate of the number of primitive operations that are executed by the algorithm. We use notations such as big-Oh to summarize the relationship between the number of operations and the input size for a problem. In this section, we demonstrate how to perform this type of running-time analysis to recursive algorithms.

With a recursive algorithm, we will account for each operation that is performed based upon the particular ***activation*** of the method that manages the flow of control at the time it is executed. Stated another way, for each invocation of the method, we only account for the number of operations that are performed within the body of that activation. We can then account for the overall number of operations that are executed as part of the recursive algorithm by taking the sum, over all activations, of the number of operations that take place during each individual activation. (As an aside, this is also the way we analyze a nonrecursive method that calls other methods from within its body.)

To demonstrate this style of analysis, we revisit the four recursive algorithms presented in Sections 5.1.1 through 5.1.4: factorial computation, drawing an English ruler, binary search, and computation of the cumulative size of a file system. In general, we may rely on the intuition afforded by a ***recursion trace*** in recognizing how many recursive activations occur, and how the parameterization of each activation can be used to estimate the number of primitive operations that occur within the body of that activation. However, each of these recursive algorithms has a unique structure and form.

Computing Factorials

It is relatively easy to analyze the efficiency of our method for computing factorials, as described in Section 5.1.1. A sample recursion trace for our factorial method was given in Figure 5.1. To compute factorial(n), we see that there are a total of $n+1$ activations, as the parameter decreases from n in the first call, to $n-1$ in the second call, and so on, until reaching the base case with parameter 0.

It is also clear, given an examination of the method body in Code Fragment 5.1, that each individual activation of factorial executes a constant number of operations. Therefore, we conclude that the overall number of operations for computing factorial(n) is $O(n)$, as there are $n+1$ activations, each of which accounts for $O(1)$ operations.

Drawing an English Ruler

In analyzing the English ruler application from Section 5.1.2, we consider the fundamental question of how many total lines of output are generated by an initial call to drawInterval(c), where c denotes the center length. This is a reasonable benchmark for the overall efficiency of the algorithm as each line of output is based upon a call to the drawLine utility, and each recursive call to drawInterval with nonzero parameter makes exactly one direct call to drawLine.

Some intuition may be gained by examining the source code and the recursion trace. We know that a call to drawInterval(c) for $c > 0$ spawns two calls to drawInterval($c-1$) and a single call to drawLine. We will rely on this intuition to prove the following claim.

Proposition 5.1: *For $c \geq 0$, a call to* drawInterval(c) *results in precisely $2^c - 1$ lines of output.*

Justification: We provide a formal proof of this claim by ***induction*** (see Section 4.4.3). In fact, induction is a natural mathematical technique for proving the correctness and efficiency of a recursive process. In the case of the ruler, we note that an application of drawInterval(0) generates no output, and that $2^0 - 1 = 1 - 1 = 0$. This serves as a base case for our claim.

More generally, the number of lines printed by drawInterval(c) is one more than twice the number generated by a call to drawInterval($c-1$), as one center line is printed between two such recursive calls. By induction, we have that the number of lines is thus $1 + 2 \cdot (2^{c-1} - 1) = 1 + 2^c - 2 = 2^c - 1$. ■

This proof is indicative of a more mathematically rigorous tool, known as a ***recurrence equation***, that can be used to analyze the running time of a recursive algorithm. That technique is discussed in Section 12.1.4, in the context of recursive sorting algorithms.

Performing a Binary Search

When considering the running time of the binary search algorithm, as presented in Section 5.1.3, we observe that a constant number of primitive operations are executed during each recursive call of the binary search method. Hence, the running time is proportional to the number of recursive calls performed. We will show that at most $\lfloor \log n \rfloor + 1$ recursive calls are made during a binary search of a sequence having n elements, leading to the following claim.

Proposition 5.2: *The binary search algorithm runs in $O(\log n)$ time for a sorted array with n elements.*

Justification: To prove this claim, a crucial fact is that with each recursive call the number of candidate elements still to be searched is given by the value

$$\mathsf{high} - \mathsf{low} + 1.$$

Moreover, the number of remaining candidates is reduced by at least one-half with each recursive call. Specifically, from the definition of mid, the number of remaining candidates is either

$$(\mathsf{mid} - 1) - \mathsf{low} + 1 = \left\lfloor \frac{\mathsf{low} + \mathsf{high}}{2} \right\rfloor - \mathsf{low} \leq \frac{\mathsf{high} - \mathsf{low} + 1}{2}$$

or

$$\mathsf{high} - (\mathsf{mid} + 1) + 1 = \mathsf{high} - \left\lfloor \frac{\mathsf{low} + \mathsf{high}}{2} \right\rfloor \leq \frac{\mathsf{high} - \mathsf{low} + 1}{2}.$$

Initially, the number of candidates is n; after the first call in a binary search, it is at most $n/2$; after the second call, it is at most $n/4$; and so on. In general, after the j^{th} call in a binary search, the number of candidate elements remaining is at most $n/2^j$. In the worst case (an unsuccessful search), the recursive calls stop when there are no more candidate elements. Hence, the maximum number of recursive calls performed, is the smallest integer r such that

$$\frac{n}{2^r} < 1.$$

In other words (recalling that we omit a logarithm's base when it is 2), r is the smallest integer such that $r > \log n$. Thus, we have

$$r = \lfloor \log n \rfloor + 1,$$

which implies that binary search runs in $O(\log n)$ time. ■

Computing Disk Space Usage

Our final recursive algorithm from Section 5.1 was that for computing the overall disk space usage in a specified portion of a file system. To characterize the "problem size" for our analysis, we let n denote the number of file-system entries in the portion of the file system that is considered. (For example, the file system portrayed in Figure 5.6 has $n = 19$ entries.)

To characterize the cumulative time spent for an initial call to diskUsage, we must analyze the total number of recursive invocations that are made, as well as the number of operations that are executed within those invocations.

We begin by showing that there are precisely n recursive invocations of the method, in particular, one for each entry in the relevant portion of the file system. Intuitively, this is because a call to diskUsage for a particular entry e of the file system is only made from within the for loop of Code Fragment 5.5 when processing the entry for the unique directory that contains e, and that entry will only be explored once.

To formalize this argument, we can define the ***nesting level*** of each entry such that the entry on which we begin has nesting level 0, entries stored directly within it have nesting level 1, entries stored within those entries have nesting level 2, and so on. We can prove by induction that there is exactly one recursive invocation of diskUsage upon each entry at nesting level k. As a base case, when $k = 0$, the only recursive invocation made is the initial one. As the inductive step, once we know there is exactly one recursive invocation for each entry at nesting level k, we can claim that there is exactly one invocation for each entry e at nesting level $k+1$, made within the for loop for the entry at level k that contains e.

Having established that there is one recursive call for each entry of the file system, we return to the question of the overall computation time for the algorithm. It would be great if we could argue that we spend $O(1)$ time in any single invocation of the method, but that is not the case. While there is a constant number of steps reflected in the call to root.length( ) to compute the disk usage directly at that entry, when the entry is a directory, the body of the diskUsage method includes a for loop that iterates over all entries that are contained within that directory. In the worst case, it is possible that one entry includes $n-1$ others.

Based on this reasoning, we could conclude that there are $O(n)$ recursive calls, each of which runs in $O(n)$ time, leading to an overall running time that is $O(n^2)$. While this upper bound is technically true, it is not a tight upper bound. Remarkably, we can prove the stronger bound that the recursive algorithm for diskUsage completes in $O(n)$ time! The weaker bound was pessimistic because it assumed a worst-case number of entries for each directory. While it is possible that some directories contain a number of entries proportional to n, they cannot all contain that many. To prove the stronger claim, we choose to consider the *overall* number of iterations of the for loop across all recursive calls. We claim there are precisely $n-1$ such iterations of that loop overall. We base this claim on the fact that each iteration of that loop makes a recursive call to diskUsage, and yet we have already concluded that there are a total of n calls to diskUsage (including the original call). We therefore conclude that there are $O(n)$ recursive calls, each of which uses $O(1)$ time outside the loop, and that the *overall* number of operations due to the loop is $O(n)$. Summing all of these bounds, the overall number of operations is $O(n)$.

The argument we have made is more advanced than with the earlier examples of recursion. The idea that we can sometimes get a tighter bound on a series of operations by considering the cumulative effect, rather than assuming that each achieves a worst case is a technique called ***amortization***; we will see another example of such analysis in Section 7.2.3. Furthermore, a file system is an implicit example of a data structure known as a ***tree***, and our disk usage algorithm is really a manifestation of a more general algorithm known as a ***tree traversal***. Trees will be the focus of Chapter 8, and our argument about the $O(n)$ running time of the disk usage algorithm will be generalized for tree traversals in Section 8.4.

5.3 Further Examples of Recursion

In this section, we provide additional examples of the use of recursion. We organize our presentation by considering the maximum number of recursive calls that may be started from within the body of a single activation.

- If a recursive call starts at most one other, we call this a ***linear recursion***.
- If a recursive call may start two others, we call this a ***binary recursion***.
- If a recursive call may start three or more others, this is ***multiple recursion***.

5.3.1 Linear Recursion

If a recursive method is designed so that each invocation of the body makes at most one new recursive call, this is know as ***linear recursion***. Of the recursions we have seen so far, the implementation of the factorial method (Section 5.1.1) is a clear example of linear recursion. More interestingly, the binary search algorithm (Section 5.1.3) is also an example of *linear recursion*, despite the term "binary" in the name. The code for binary search (Code Fragment 5.3) includes a case analysis, with two branches that lead to a further recursive call, but only one branch is followed during a particular execution of the body.

A consequence of the definition of linear recursion is that any recursion trace will appear as a single sequence of calls, as we originally portrayed for the factorial method in Figure 5.1 of Section 5.1.1. Note that the *linear recursion* terminology reflects the structure of the recursion trace, not the asymptotic analysis of the running time; for example, we have seen that binary search runs in $O(\log n)$ time.

Summing the Elements of an Array Recursively

Linear recursion can be a useful tool for processing a sequence, such as a Java array. Suppose, for example, that we want to compute the sum of an array of n integers. We can solve this summation problem using linear recursion by observing that if $n = 0$ the sum is trivially 0, and otherwise it is the sum of the first $n-1$ integers in the array plus the last value in the array. (See Figure 5.9.)

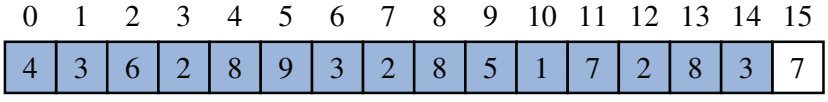


Figure 5.9: Computing the sum of a sequence recursively, by adding the last number to the sum of the first $n-1$.

A recursive algorithm for computing the sum of an array of integers based on this intuition is implemented in Code Fragment 5.6.

```
1  /** Returns the sum of the first n integers of the given array. */
2  public static int linearSum(int[ ] data, int n) {
3    if (n == 0)
4      return 0;
5    else
6      return linearSum(data, n−1) + data[n−1];
7  }
```

Code Fragment 5.6: Summing an array of integers using linear recursion.

A recursion trace of the linearSum method for a small example is given in Figure 5.10. For an input of size n, the linearSum algorithm makes $n+1$ method calls. Hence, it will take $O(n)$ time, because it spends a constant amount of time performing the nonrecursive part of each call. Moreover, we can also see that the memory space used by the algorithm (in addition to the array) is also $O(n)$, as we use a constant amount of memory space for each of the $n+1$ frames in the trace at the time we make the final recursive call (with $n = 0$).

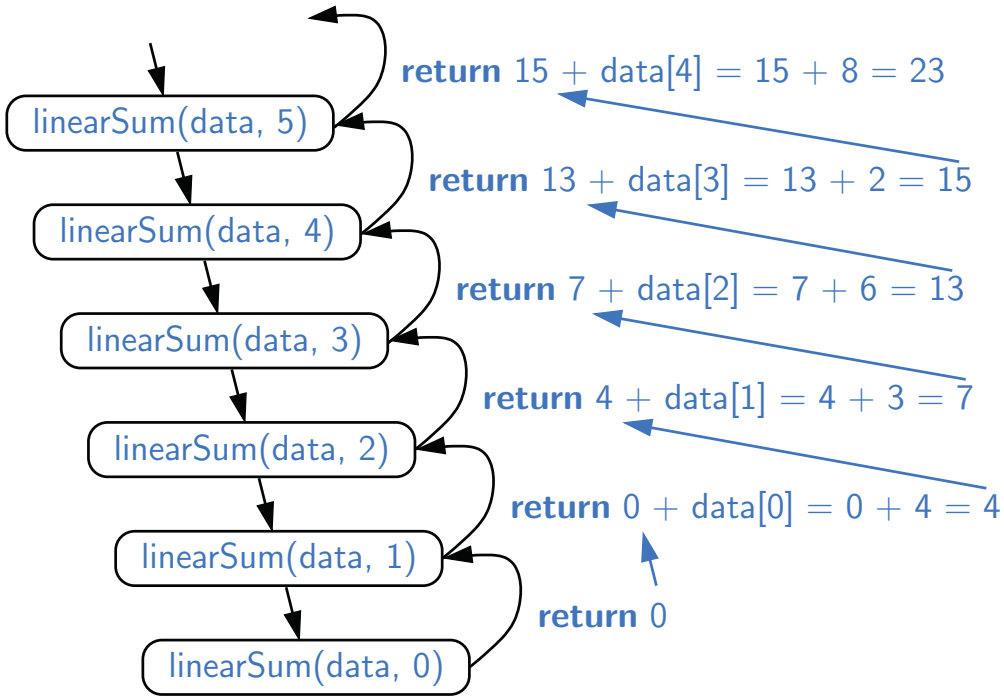


Figure 5.10: Recursion trace for an execution of linearSum(data, 5) with input parameter data = 4, 3, 6, 2, 8.

Reversing a Sequence with Recursion

Next, let us consider the problem of reversing the n elements of an array, so that the first element becomes the last, the second element becomes second to the last, and so on. We can solve this problem using linear recursion, by observing that the reversal of a sequence can be achieved by swapping the first and last elements and then recursively reversing the remaining elements. We present an implementation of this algorithm in Code Fragment 5.7, using the convention that the first time we call this algorithm we do so as reverseArray(data, 0, n−1).

```
1  /** Reverses the contents of subarray data[low] through data[high] inclusive. */
2  public static void reverseArray(int[ ] data, int low, int high) {
3    if (low < high) {                          // if at least two elements in subarray
4      int temp = data[low];                    // swap data[low] and data[high]
5      data[low] = data[high];
6      data[high] = temp;
7      reverseArray(data, low + 1, high − 1);   // recur on the rest
8    }
9  }
```

Code Fragment 5.7: Reversing the elements of an array using linear recursion.

We note that whenever a recursive call is made, there will be two fewer elements in the relevant portion of the array. (See Figure 5.11.) Eventually a base case is reached when the condition low < high fails, either because low == high in the case that n is odd, or because low == high + 1 in the case that n is even.

The above argument implies that the recursive algorithm of Code Fragment 5.7 is guaranteed to terminate after a total of $1 + \lfloor \frac{n}{2} \rfloor$ recursive calls. Because each call involves a constant amount of work, the entire process runs in $O(n)$ time.

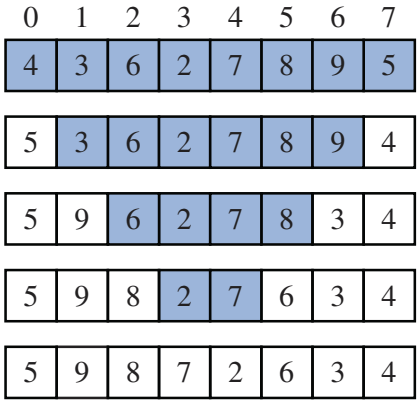


Figure 5.11: A trace of the recursion for reversing a sequence. The highlighted portion has yet to be reversed.

Recursive Algorithms for Computing Powers

As another interesting example of the use of linear recursion, we consider the problem of raising a number x to an arbitrary nonnegative integer n. That is, we wish to compute the ***power function***, defined as $power(x,n) = x^n$. (We use the name "power" for this discussion, to differentiate from the pow method of the Math class, which provides such functionality.) We will consider two different recursive formulations for the problem that lead to algorithms with very different performance.

A trivial recursive definition follows from the fact that $x^n = x \cdot x^{n-1}$ for $n > 0$.

$$power(x,n) = \begin{cases} 1 & \text{if } n = 0 \\ x \cdot power(x, n-1) & \text{otherwise.} \end{cases}$$

This definition leads to a recursive algorithm shown in Code Fragment 5.8.

```
1  /** Computes the value of x raised to the nth power, for nonnegative integer n. */
2  public static double power(double x, int n) {
3    if (n == 0)
4      return 1;
5    else
6      return x * power(x, n-1);
7  }
```

Code Fragment 5.8: Computing the power function using trivial recursion.

A recursive call to this version of $power(x,n)$ runs in $O(n)$ time. Its recursion trace has structure very similar to that of the factorial function from Figure 5.1, with the parameter decreasing by one with each call, and constant work performed at each of $n+1$ levels.

However, there is a much faster way to compute the power function using an alternative definition that employs a squaring technique. Let $k = \lfloor \frac{n}{2} \rfloor$ denote the floor of the integer division (equivalent to n/2 in Java when n is an int). We consider the expression $\left(x^k\right)^2$. When n is even, $\lfloor \frac{n}{2} \rfloor = \frac{n}{2}$ and therefore $\left(x^k\right)^2 = \left(x^{\frac{n}{2}}\right)^2 = x^n$. When n is odd, $\lfloor \frac{n}{2} \rfloor = \frac{n-1}{2}$ and $\left(x^k\right)^2 = x^{n-1}$, and therefore $x^n = \left(x^k\right)^2 \cdot x$, just as $2^{13} = \left(2^6 \cdot 2^6\right) \cdot 2$. This analysis leads to the following recursive definition:

$$power(x,n) = \begin{cases} 1 & \text{if } n = 0 \\ \left(power\left(x, \lfloor \frac{n}{2} \rfloor\right)\right)^2 \cdot x & \text{if } n > 0 \text{ is odd} \\ \left(power\left(x, \lfloor \frac{n}{2} \rfloor\right)\right)^2 & \text{if } n > 0 \text{ is even} \end{cases}$$

If we were to implement this recursion making *two* recursive calls to compute $power(x, \lfloor \frac{n}{2} \rfloor) \cdot power(x, \lfloor \frac{n}{2} \rfloor)$, a trace of the recursion would demonstrate $O(n)$ calls. We can perform significantly fewer operations by computing $power(x, \lfloor \frac{n}{2} \rfloor)$ and storing it in a variable as a partial result, and then multiplying it by itself. An implementation based on this recursive definition is given in Code Fragment 5.9.

```
1  /** Computes the value of x raised to the nth power, for nonnegative integer n. */
2  public static double power(double x, int n) {
3    if (n == 0)
4      return 1;
5    else {
6      double partial = power(x, n/2);          // rely on truncated division of n
7      double result = partial * partial;
8      if (n % 2 == 1)                          // if n odd, include extra factor of x
9        result *= x;
10     return result;
11   }
12 }
```

Code Fragment 5.9: Computing the power function using repeated squaring.

To illustrate the execution of our improved algorithm, Figure 5.12 provides a recursion trace of the computation power(2, 13).

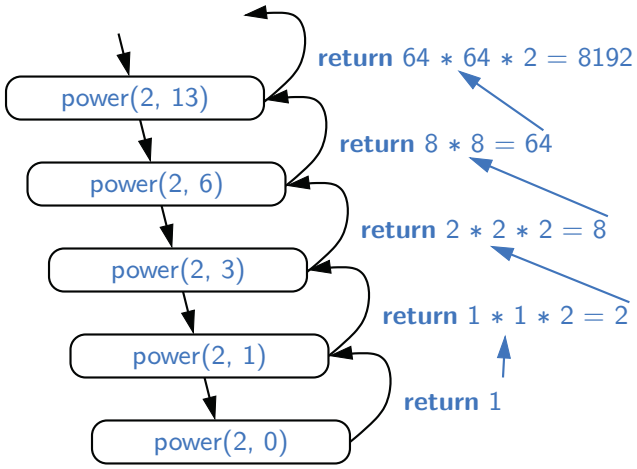


Figure 5.12: Recursion trace for an execution of power(2, 13).

To analyze the running time of the revised algorithm, we observe that the exponent in each recursive call of method power(x,n) is at most half of the preceding exponent. As we saw with the analysis of binary search, the number of times that we can divide n by two before getting to one or less is $O(\log n)$. Therefore, our new formulation of power results in $O(\log n)$ recursive calls. Each individual activation of the method uses $O(1)$ operations (excluding the recursive call), and so the total number of operations for computing power(x,n) is $O(\log n)$. This is a significant improvement over the original $O(n)$-time algorithm.

The improved version also provides significant saving in reducing the memory usage. The first version has a recursive depth of $O(n)$, and therefore, $O(n)$ frames are simultaneously stored in memory. Because the recursive depth of the improved version is $O(\log n)$, its memory usage is $O(\log n)$ as well.

5.3.2 Binary Recursion

When a method makes two recursive calls, we say that it uses ***binary recursion***. We have already seen an example of binary recursion when drawing the English ruler (Section 5.1.2). As another application of binary recursion, let us revisit the problem of summing the n integers of an array. Computing the sum of one or zero values is trivial. With two or more values, we can recursively compute the sum of the first half, and the sum of the second half, and add those sums together. Our implementation of such an algorithm, in Code Fragment 5.10, is initially invoked as binarySum(data, 0, n−1).

```
1  /** Returns the sum of subarray data[low] through data[high] inclusive. */
2  public static int binarySum(int[ ] data, int low, int high) {
3    if (low > high)                                    // zero elements in subarray
4      return 0;
5    else if (low == high)                              // one element in subarray
6      return data[low];
7    else {
8      int mid = (low + high) / 2;
9      return binarySum(data, low, mid) + binarySum(data, mid+1, high);
10   }
11 }
```

Code Fragment 5.10: Summing the elements of a sequence using binary recursion.

To analyze algorithm binarySum, we consider, for simplicity, the case where n is a power of two. Figure 5.13 shows the recursion trace of an execution of binarySum(data, 0, 7). We label each box with the values of parameters low and high for that call. The size of the range is divided in half at each recursive call, and so the depth of the recursion is $1+\log_2 n$. Therefore, binarySum uses $O(\log n)$ amount of additional space, which is a big improvement over the $O(n)$ space used by the linearSum method of Code Fragment 5.6. However, the running time of binarySum is $O(n)$, as there are $2n-1$ method calls, each requiring constant time.

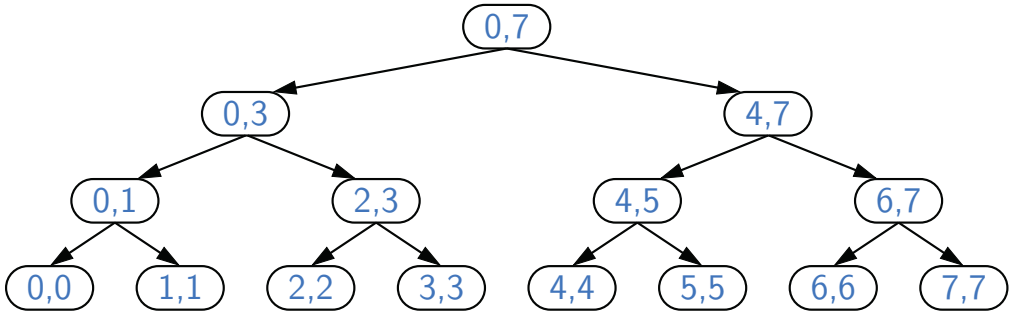


Figure 5.13: Recursion trace for the execution of binarySum(data, 0, 7).

5.3.3 Multiple Recursion

Generalizing from binary recursion, we define ***multiple recursion*** as a process in which a method may make more than two recursive calls. Our recursion for analyzing the disk space usage of a file system (see Section 5.1.4) is an example of multiple recursion, because the number of recursive calls made during one invocation was equal to the number of entries within a given directory of the file system.

Another common application of multiple recursion is when we want to enumerate various configurations in order to solve a combinatorial puzzle. For example, the following are all instances of what are known as ***summation puzzles***:

$$\begin{aligned} pot + pan &= bib \\ dog + cat &= pig \\ boy + girl &= baby \end{aligned}$$

To solve such a puzzle, we need to assign a unique digit (that is, $0, 1, \ldots, 9$) to each letter in the equation, in order to make the equation true. Typically, we solve such a puzzle by using our human observations of the particular puzzle we are trying to solve to eliminate configurations (that is, possible partial assignments of digits to letters) until we can work through the feasible configurations that remain, testing for the correctness of each one.

If the number of possible configurations is not too large, however, we can use a computer to simply enumerate all the possibilities and test each one, without employing any human observations. Such an algorithm can use multiple recursion to work through the configurations in a systematic way. To keep the description general enough to be used with other puzzles, we consider an algorithm that enumerates and tests all k-length sequences, without repetitions, chosen from a given universe U. We show pseudocode for such an algorithm in Code Fragment 5.11, building the sequence of k elements with the following steps:

1. Recursively generating the sequences of $k-1$ elements
2. Appending to each such sequence an element not already contained in it.

Throughout the execution of the algorithm, we use a set U to keep track of the elements not contained in the current sequence, so that an element e has not been used yet if and only if e is in U.

Another way to look at the algorithm of Code Fragment 5.11 is that it enumerates every possible size-k ordered subset of U, and tests each subset for being a possible solution to our puzzle.

For summation puzzles, $U = \{0,1,2,3,4,5,6,7,8,9\}$ and each position in the sequence corresponds to a given letter. For example, the first position could stand for b, the second for o, the third for y, and so on.

Algorithm PuzzleSolve(k, S, U):
Input: An integer k, sequence S, and set U
Output: An enumeration of all k-length extensions to S using elements in U without repetitions

```
for each e in U do
    Add e to the end of S
    Remove e from U                                  {e is now being used}
    if k == 1 then
        Test whether S is a configuration that solves the puzzle
        if S solves the puzzle then
            add S to output                          {a solution}
    else
        PuzzleSolve(k − 1, S, U)                     {a recursive call}
    Remove e from the end of S
    Add e back to U                                  {e is now considered as unused}
```

Code Fragment 5.11: Solving a combinatorial puzzle by enumerating and testing all possible configurations.

In Figure 5.14, we show a recursion trace of a call to PuzzleSolve(3, S, U), where S is empty and $U = \{a,b,c\}$. During the execution, all the permutations of the three characters are generated and tested. Note that the initial call makes three recursive calls, each of which in turn makes two more. If we had executed PuzzleSolve(3, S, U) on a set U consisting of four elements, the initial call would have made four recursive calls, each of which would have a trace looking like the one in Figure 5.14.

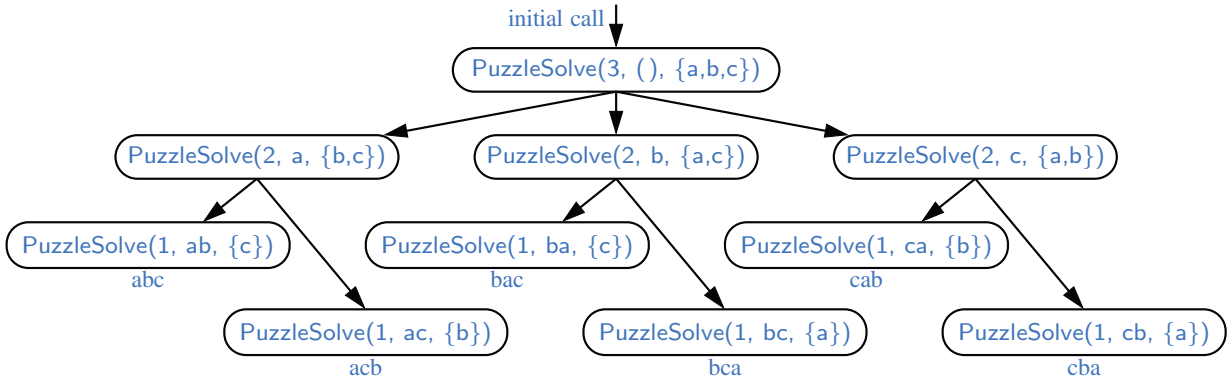


Figure 5.14: Recursion trace for an execution of PuzzleSolve(3, S, U), where S is empty and $U = \{a,b,c\}$. This execution generates and tests all permutations of a, b, and c. We show the permutations generated directly below their respective boxes.

5.4 Designing Recursive Algorithms

An algorithm that uses recursion typically has the following form:

- ***Test for base cases.*** We begin by testing for a set of base cases (there should be at least one). These base cases should be defined so that every possible chain of recursive calls will eventually reach a base case, and the handling of each base case should not use recursion.
- ***Recur.*** If not a base case, we perform one or more recursive calls. This recursive step may involve a test that decides which of several possible recursive calls to make. We should define each possible recursive call so that it makes progress towards a base case.

Parameterizing a Recursion

To design a recursive algorithm for a given problem, it is useful to think of the different ways we might define subproblems that have the same general structure as the original problem. If one has difficulty finding the repetitive structure needed to design a recursive algorithm, it is sometimes useful to work out the problem on a few concrete examples to see how the subproblems should be defined.

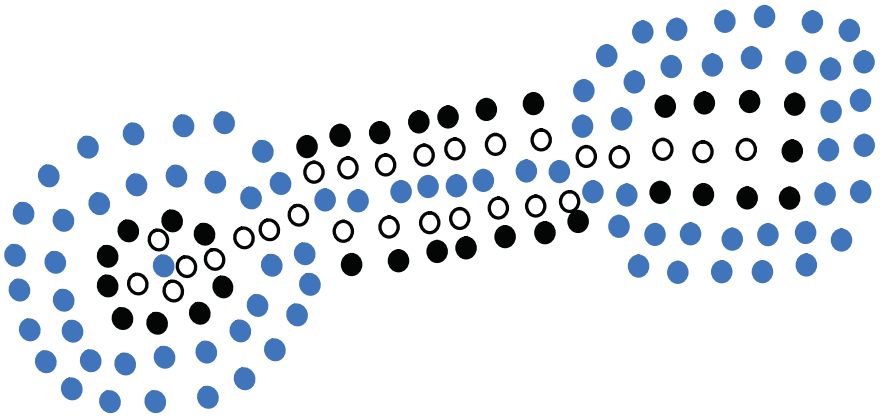
A successful recursive design sometimes requires that we redefine the original problem to facilitate similar-looking subproblems. Often, this involved reparameterizing the signature of the method. For example, when performing a binary search in an array, a natural method signature for a caller would appear as binarySearch(data, target). However, in Section 5.1.3, we defined our method with calling signature binarySearch(data, target, low, high), using the additional parameters to demarcate subarrays as the recursion proceeds. This change in parameterization is critical for binary search. Several other examples in this chapter (e.g., reverseArray, linearSum, binarySum) also demonstrated the use of additional parameters in defining recursive subproblems.

If we wish to provide a cleaner public interface to an algorithm without exposing the user to the recursive parameterization, a standard technique is to make the recursive version private, and to introduce a cleaner public method (that calls the private one with appropriate parameters). For example, we might offer the following simpler version of binarySearch for public use:

```
/** Returns true if the target value is found in the data array. */
public static boolean binarySearch(int[ ] data, int target) {
  return binarySearch(data, target, 0, data.length − 1);  // use parameterized version
}
```

Chapter 6

Stacks, Queues, and Deques



Contents

6.1 Stacks

A ***stack*** is a collection of objects that are inserted and removed according to the ***last-in, first-out*** (***LIFO***) principle. A user may insert objects into a stack at any time, but may only access or remove the most recently inserted object that remains (at the so-called "top" of the stack). The name "stack" is derived from the metaphor of a stack of plates in a spring-loaded, cafeteria plate dispenser. In this case, the fundamental operations involve the "pushing" and "popping" of plates on the stack. When we need a new plate from the dispenser, we "pop" the top plate off the stack, and when we add a plate, we "push" it down on the stack to become the new top plate. Perhaps an even more amusing example is a PEZ® candy dispenser, which stores mint candies in a spring-loaded container that "pops" out the topmost candy in the stack when the top of the dispenser is lifted (see Figure 6.1).

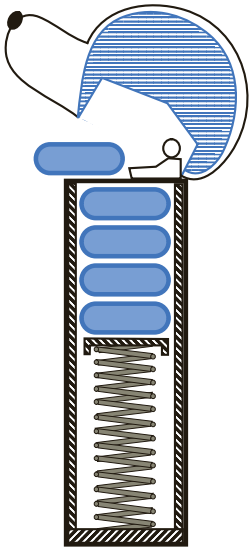


Figure 6.1: A schematic drawing of a PEZ® dispenser; a physical implementation of the stack ADT. (PEZ® is a registered trademark of PEZ Candy, Inc.)

Stacks are a fundamental data structure. They are used in many applications, including the following.

Example 6.1: *Internet Web browsers store the addresses of recently visited sites on a stack. Each time a user visits a new site, that site's address is "pushed" onto the stack of addresses. The browser then allows the user to "pop" back to previously visited sites using the "back" button.*

Example 6.2: *Text editors usually provide an "undo" mechanism that cancels recent editing operations and reverts to former states of a document. This undo operation can be accomplished by keeping text changes in a stack.*

6.1.1 The Stack Abstract Data Type

Stacks are the simplest of all data structures, yet they are also among the most important, as they are used in a host of different applications, and as a tool for many more sophisticated data structures and algorithms. Formally, a stack is an abstract data type (ADT) that supports the following two update methods:

push(e): Adds element e to the top of the stack.

pop( ): Removes and returns the top element from the stack (or null if the stack is empty).

Additionally, a stack supports the following accessor methods for convenience:

top( ): Returns the top element of the stack, without removing it (or null if the stack is empty).

size( ): Returns the number of elements in the stack.

isEmpty( ): Returns a boolean indicating whether the stack is empty.

By convention, we assume that elements added to the stack can have arbitrary type and that a newly created stack is empty.

Example 6.3: *The following table shows a series of stack operations and their effects on an initially empty stack S of integers.*

Method	Return Value	Stack Contents
push(5)	–	(5)
push(3)	–	(5, 3)
size( )	2	(5, 3)
pop( )	3	(5)
isEmpty( )	false	(5)
pop( )	5	( )
isEmpty( )	true	( )
pop( )	null	( )
push(7)	–	(7)
push(9)	–	(7, 9)
top( )	9	(7, 9)
push(4)	–	(7, 9, 4)
size( )	3	(7, 9, 4)
pop( )	4	(7, 9)
push(6)	–	(7, 9, 6)
push(8)	–	(7, 9, 6, 8)
pop( )	8	(7, 9, 6)

A Stack Interface in Java

In order to formalize our abstraction of a stack, we define what is known as its ***application programming interface*** (API) in the form of a Java ***interface***, which describes the names of the methods that the ADT supports and how they are to be declared and used. This interface is defined in Code Fragment 6.1.

We rely on Java's ***generics framework*** (described in Section 2.5.2), allowing the elements stored in the stack to belong to any object type <E>. For example, a variable representing a stack of integers could be declared with type Stack<Integer>. The formal type parameter is used as the parameter type for the push method, and the return type for both pop and top.

Recall, from the discussion of Java interfaces in Section 2.3.1, that the interface serves as a type definition but that it cannot be directly instantiated. For the ADT to be of any use, we must provide one or more concrete classes that implement the methods of the interface associated with that ADT. In the following subsections, we will give two such implementations of the Stack interface: one that uses an array for storage and another that uses a linked list.

The java.util.Stack Class

Because of the importance of the stack ADT, Java has included, since its original version, a concrete class named java.util.Stack that implements the LIFO semantics of a stack. However, Java's Stack class remains only for historic reasons, and its interface is not consistent with most other data structures in the Java library. In fact, the current documentation for the Stack class recommends that it not be used, as LIFO functionality (and more) is provided by a more general data structure known as a double-ended queue (which we describe in Section 6.3).

For the sake of comparison, Table 6.1 provides a side-by-side comparison of the interface for our stack ADT and the `java.util.Stack` class. In addition to some differences in method names, we note that methods pop and peek of the java.util.Stack class throw a custom EmptyStackException if called when the stack is empty (whereas null is returned in our abstraction).

Our Stack ADT	Class java.util.Stack	
size( )	size( )	
isEmpty( )	empty( )	⇐
push(e)	push(e)	
pop( )	pop( )	
top( )	peek( )	⇐

Table 6.1: Methods of our stack ADT and corresponding methods of the class java.util.Stack, with differences highlighted in the right margin.

```
1  /**
2   * A collection of objects that are inserted and removed according to the last-in
3   * first-out principle. Although similar in purpose, this interface differs from
4   * java.util.Stack.
5   *
6   * @author Michael T. Goodrich
7   * @author Roberto Tamassia
8   * @author Michael H. Goldwasser
9   */
10 public interface Stack<E> {
11
12   /**
13    * Returns the number of elements in the stack.
14    * @return number of elements in the stack
15    */
16   int size( );
17
18   /**
19    * Tests whether the stack is empty.
20    * @return true if the stack is empty, false otherwise
21    */
22   boolean isEmpty( );
23
24   /**
25    * Inserts an element at the top of the stack.
26    * @param e   the element to be inserted
27    */
28   void push(E e);
29
30   /**
31    * Returns, but does not remove, the element at the top of the stack.
32    * @return top element in the stack (or null if empty)
33    */
34   E top( );
35
36   /**
37    * Removes and returns the top element from the stack.
38    * @return element removed (or null if empty)
39    */
40   E pop( );
41 }
```

Code Fragment 6.1: Interface Stack documented with comments in Javadoc style (Section 1.9.4). Note also the use of the generic parameterized type, E, which allows a stack to contain elements of any specified (reference) type.

6.1.2 A Simple Array-Based Stack Implementation

As our first implementation of the stack ADT, we store elements in an array, named data, with capacity N for some fixed N. We oriented the stack so that the bottom element of the stack is always stored in cell data[0], and the top element of the stack in cell data[t] for index t that is equal to one less than the current size of the stack. (See Figure 6.2.)

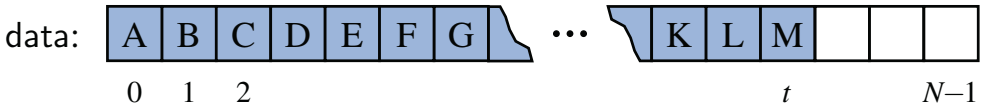


Figure 6.2: Representing a stack with an array; the top element is in cell data[t].

Recalling that arrays start at index 0 in Java, when the stack holds elements from data[0] to data[t] inclusive, it has size $t+1$. By convention, when the stack is empty it will have t equal to -1 (and thus has size $t+1$, which is 0). A complete Java implementation based on this strategy is given in Code Fragment 6.2 (with Javadoc comments omitted due to space considerations).

```
1  public class ArrayStack<E> implements Stack<E> {
2    public static final int CAPACITY=1000;  // default array capacity
3    private E[ ] data;                      // generic array used for storage
4    private int t = -1;                     // index of the top element in stack
5    public ArrayStack( ) { this(CAPACITY); } // constructs stack with default capacity
6    public ArrayStack(int capacity) {       // constructs stack with given capacity
7      data = (E[ ]) new Object[capacity];   // safe cast; compiler may give warning
8    }
9    public int size( ) { return (t + 1); }
10   public boolean isEmpty( ) { return (t == -1); }
11   public void push(E e) throws IllegalStateException {
12     if (size( ) == data.length) throw new IllegalStateException("Stack is full");
13     data[++t] = e;                        // increment t before storing new item
14   }
15   public E top( ) {
16     if (isEmpty( )) return null;
17     return data[t];
18   }
19   public E pop( ) {
20     if (isEmpty( )) return null;
21     E answer = data[t];
22     data[t] = null;                       // dereference to help garbage collection
23     t--;
24     return answer;
25   }
26 }
```

Code Fragment 6.2: Array-based implementation of the Stack interface.

A Drawback of This Array-Based Stack Implementation

The array implementation of a stack is simple and efficient. Nevertheless, this implementation has one negative aspect—it relies on a fixed-capacity array, which limits the ultimate size of the stack.

For convenience, we allow the user of a stack to specify the capacity as a parameter to the constructor (and offer a default constructor that uses capacity of 1,000). In cases where a user has a good estimate on the number of items needing to go in the stack, the array-based implementation is hard to beat. However, if the estimate is wrong, there can be grave consequences. If the application needs much less space than the reserved capacity, memory is wasted. Worse yet, if an attempt is made to push an item onto a stack that has already reached its maximum capacity, the implementation of Code Fragment 6.2 throws an IllegalStateException, refusing to store the new element. Thus, even with its simplicity and efficiency, the array-based stack implementation is not necessarily ideal.

Fortunately, we will later demonstrate two approaches for implementing a stack without such a size limitation and with space always proportional to the actual number of elements stored in the stack. One approach, given in the next subsection uses a singly linked list for storage; in Section 7.2.1, we will provide a more advanced array-based approach that overcomes the limit of a fixed capacity.

Analyzing the Array-Based Stack Implementation

The correctness of the methods in the array-based implementation follows from our definition of index t. Note well that when pushing an element, t is incremented before placing the new element, so that it uses the first available cell.

Table 6.2 shows the running times for methods of this array-based stack implementation. Each method executes a constant number of statements involving arithmetic operations, comparisons, and assignments, or calls to size and isEmpty, which both run in constant time. Thus, in this implementation of the stack ADT, each method runs in constant time, that is, they each run in $O(1)$ time.

Method	Running Time
size	$O(1)$
isEmpty	$O(1)$
top	$O(1)$
push	$O(1)$
pop	$O(1)$

Table 6.2: Performance of a stack realized by an array. The space usage is $O(N)$, where N is the size of the array, determined at the time the stack is instantiated, and independent from the number $n \leq N$ of elements that are actually in the stack.

Garbage Collection in Java

We wish to draw attention to one interesting aspect involving the implementation of the pop method in Code Fragment 6.2. We set a local variable, answer, to reference the element that is being popped, and then we intentionally reset data[t] to **null** at line 22, before decrementing t. The assignment to **null** was not technically required, as our stack would still operate correctly without it.

Our reason for returning the cell to a null reference is to assist Java's ***garbage collection*** mechanism, which searches memory for objects that are no longer actively referenced and reclaims their space for future use. (For more details, see Section 15.1.3.) If we continued to store a reference to the popped element in our array, the stack class would ignore it (eventually overwriting the reference if more elements get added to the stack). But, if there were no other active references to the element in the user's application, that spurious reference in the stack's array would stop Java's garbage collector from reclaiming the element.

Sample Usage

We conclude this section by providing a demonstration of code that creates and uses an instance of the ArrayStack class. In this example, we declare the parameterized type of the stack as the Integer wrapper class. This causes the signature of the push method to accept an Integer instance as a parameter, and for the return type of both top and pop to be an Integer. Of course, with Java's autoboxing and unboxing (see Section 1.3), a primitive **int** can be sent as a parameter to push.

```
Stack<Integer> S = new ArrayStack<>();      // contents: ()
S.push(5);                                  // contents: (5)
S.push(3);                                  // contents: (5, 3)
System.out.println(S.size());               // contents: (5, 3)      outputs 2
System.out.println(S.pop());                // contents: (5)         outputs 3
System.out.println(S.isEmpty());            // contents: (5)         outputs false
System.out.println(S.pop());                // contents: ()          outputs 5
System.out.println(S.isEmpty());            // contents: ()          outputs true
System.out.println(S.pop());                // contents: ()          outputs null
S.push(7);                                  // contents: (7)
S.push(9);                                  // contents: (7, 9)
System.out.println(S.top());                // contents: (7, 9)      outputs 9
S.push(4);                                  // contents: (7, 9, 4)
System.out.println(S.size());               // contents: (7, 9, 4)   outputs 3
System.out.println(S.pop());                // contents: (7, 9)      outputs 4
S.push(6);                                  // contents: (7, 9, 6)
S.push(8);                                  // contents: (7, 9, 6, 8)
System.out.println(S.pop());                // contents: (7, 9, 6)   outputs 8
```

Code Fragment 6.3: Sample usage of our ArrayStack class.

6.1.3 Implementing a Stack with a Singly Linked List

In this section, we demonstrate how the Stack interface can be easily implemented using a singly linked list for storage. Unlike our array-based implementation, the linked-list approach has memory usage that is always proportional to the number of actual elements currently in the stack, and without an arbitrary capacity limit.

In designing such an implementation, we need to decide if the top of the stack is at the front or back of the list. There is clearly a best choice here, however, since we can insert and delete elements in constant time only at the front. With the top of the stack stored at the front of the list, all methods execute in constant time.

The Adapter Pattern

The ***adapter*** design pattern applies to any context where we effectively want to modify an existing class so that its methods match those of a related, but different, class or interface. One general way to apply the adapter pattern is to define a new class in such a way that it contains an instance of the existing class as a hidden field, and then to implement each method of the new class using methods of this hidden instance variable. By applying the adapter pattern in this way, we have created a new class that performs some of the same functions as an existing class, but repackaged in a more convenient way.

In the context of the stack ADT, we can adapt our SinglyLinkedList class of Section 3.2.1 to define a new LinkedStack class, shown in Code Fragment 6.4. This class declares a SinglyLinkedList named list as a private field, and uses the following correspondences:

Stack Method	*Singly Linked List Method*
size( )	list.size( )
isEmpty( )	list.isEmpty( )
push(e)	list.addFirst(e)
pop( )	list.removeFirst( )
top( )	list.first( )

```
1  public class LinkedStack<E> implements Stack<E> {
2    private SinglyLinkedList<E> list = new SinglyLinkedList<>();    // an empty list
3    public LinkedStack( ) { }                  // new stack relies on the initially empty list
4    public int size( ) { return list.size( ); }
5    public boolean isEmpty( ) { return list.isEmpty( ); }
6    public void push(E element) { list.addFirst(element); }
7    public E top( ) { return list.first( ); }
8    public E pop( ) { return list.removeFirst( ); }
9  }
```

Code Fragment 6.4: Implementation of a Stack using a SinglyLinkedList as storage.

6.1.4 Reversing an Array Using a Stack

As a consequence of the LIFO protocol, a stack can be used as a general toll to reverse a data sequence. For example, if the values 1, 2, and 3 are pushed onto a stack in that order, they will be popped from the stack in the order 3, 2, and then 1.

We demonstrate this concept by revisiting the problem of reversing the elements of an array. (We provided a recursive algorithm for this task in Section 5.3.1.) We create an empty stack for auxiliary storage, push all of the array elements onto the stack, and then pop those elements off of the stack while overwriting the cells of the array from beginning to end. In Code Fragment 6.5, we give a Java implementation of this algorithm. We show an example use of this method in Code Fragment 6.6.

```
1  /** A generic method for reversing an array. */
2  public static <E> void reverse(E[ ] a) {
3    Stack<E> buffer = new ArrayStack<>(a.length);
4    for (int i=0; i < a.length; i++)
5      buffer.push(a[i]);
6    for (int i=0; i < a.length; i++)
7      a[i] = buffer.pop( );
8  }
```

Code Fragment 6.5: A generic method that reverses the elements in an array with objects of type E, using a stack declared with the interface Stack<E> as its type.

```
1  /** Tester routine for reversing arrays */
2  public static void main(String args[ ]) {
3    Integer[ ] a = {4, 8, 15, 16, 23, 42};      // autoboxing allows this
4    String[ ] s = {"Jack", "Kate", "Hurley", "Jin", "Michael"};
5    System.out.println("a = " + Arrays.toString(a));
6    System.out.println("s = " + Arrays.toString(s));
7    System.out.println("Reversing...");
8    reverse(a);
9    reverse(s);
10   System.out.println("a = " + Arrays.toString(a));
11   System.out.println("s = " + Arrays.toString(s));
12 }
```

The output from this method is the following:

```
a = [4, 8, 15, 16, 23, 42]
s = [Jack, Kate, Hurley, Jin, Michael]
Reversing...
a = [42, 23, 16, 15, 8, 4]
s = [Michael, Jin, Hurley, Kate, Jack]
```

Code Fragment 6.6: A test of the reverse method using two arrays.

6.1.5 Matching Parentheses and HTML Tags

In this subsection, we explore two related applications of stacks, both of which involve testing for pairs of matching delimiters. In our first application, we consider arithmetic expressions that may contain various pairs of grouping symbols, such as

- Parentheses: "(" and ")"
- Braces: "{" and "}"
- Brackets: "[" and "]"

Each opening symbol must match its corresponding closing symbol. For example, a left bracket, "[," must match a corresponding right bracket, "]," as in the following expression

$$[(5+x)-(y+z)].$$

The following examples further illustrate this concept:

- Correct: ()(()){([()])}
- Correct: ((()(()){([()])}))
- Incorrect:)(()){([()])}
- Incorrect: ({[])}
- Incorrect: (

We leave the precise definition of a matching group of symbols to Exercise R-6.6.

An Algorithm for Matching Delimiters

An important task when processing arithmetic expressions is to make sure their delimiting symbols match up correctly. We can use a stack to perform this task with a single left-to-right scan of the original string.

Each time we encounter an opening symbol, we push that symbol onto the stack, and each time we encounter a closing symbol, we pop a symbol from the stack (assuming it is not empty) and check that these two symbols form a valid pair. If we reach the end of the expression and the stack is empty, then the original expression was properly matched. Otherwise, there must be an opening delimiter on the stack without a matching symbol. If the length of the original expression is n, the algorithm will make at most n calls to push and n calls to pop. Code Fragment 6.7 presents a Java implementation of such an algorithm. It specifically checks for delimiter pairs ( ), { }, and [ ], but could easily be changed to accommodate further symbols. Specifically, we define two fixed strings, `"({["` and `")}]"`, that are intentionally coordinated to reflect the symbol pairs. When examining a character of the expression string, we call the indexOf method of the String class on these special strings to determine if the character matches a delimiter and, if so, which one. Method indexOf returns the the index at which a given character is first found in a string (or -1 if the character is not found).

```
1  /** Tests if delimiters in the given expression are properly matched. */
2  public static boolean isMatched(String expression) {
3    final String opening  = "({[";          // opening delimiters
4    final String closing  = ")}]";          // respective closing delimiters
5    Stack<Character> buffer = new LinkedStack<>();
6    for (char c : expression.toCharArray()) {
7      if (opening.indexOf(c) != -1)         // this is a left delimiter
8        buffer.push(c);
9      else if (closing.indexOf(c) != -1) {  // this is a right delimiter
10       if (buffer.isEmpty())               // nothing to match with
11         return false;
12       if (closing.indexOf(c) != opening.indexOf(buffer.pop()))
13         return false;                     // mismatched delimiter
14     }
15   }
16   return buffer.isEmpty();                // were all opening delimiters matched?
17 }
```

Code Fragment 6.7: Method for matching delimiters in an arithmetic expression.

Matching Tags in a Markup Language

Another application of matching delimiters is in the validation of markup languages such as HTML or XML. HTML is the standard format for hyperlinked documents on the Internet and XML is an extensible markup language used for a variety of structured data sets. We show a sample HTML document in Figure 6.3.

```
<body>
<center>
<h1> The Little Boat </h1>
</center>
<p> The storm tossed the little
boat like a cheap sneaker in an
old washing machine.  The three
drunken fishermen were used to
such treatment, of course, but
not the tree salesman, who even as
a stowaway now felt that he
had overpaid for the voyage. </p>
<ol>
<li> Will the salesman die? </li>
<li> What color is the boat? </li>
<li> And what about Naomi? </li>
</ol>
</body>
```

(a)

The Little Boat

The storm tossed the little boat like a cheap sneaker in an old washing machine. The three drunken fishermen were used to such treatment, of course, but not the tree salesman, who even as a stowaway now felt that he had overpaid for the voyage.

1. Will the salesman die?
2. What color is the boat?
3. And what about Naomi?

(b)

Figure 6.3: Illustrating (a) an HTML document and (b) its rendering.

In an HTML document, portions of text are delimited by ***HTML tags***. A simple opening HTML tag has the form "<name>" and the corresponding closing tag has the form "</name>". For example, we see the <body> tag on the first line of Figure 6.3a, and the matching </body> tag at the close of that document. Other commonly used HTML tags that are used in this example include:

- <body>: document body
- <h1>: section header
- : center justify
- <p>: paragraph
- : numbered (ordered) list
- : list item

Ideally, an HTML document should have matching tags, although most browsers tolerate a certain number of mismatching tags. In Code Fragment 6.8, we give a Java method that matches tags in a string representing an HTML document.

We make a left-to-right pass through the raw string, using index j to track our progress. The indexOf method of the String class, which optionally accepts a starting index as a second parameter, locates the '<' and '>' characters that define the tags. Method substring, also of the String class, returns the substring starting at a given index and optionally ending right before another given index. Opening tags are pushed onto the stack, and matched against closing tags as they are popped from the stack, just as we did when matching delimiters in Code Fragment 6.7.

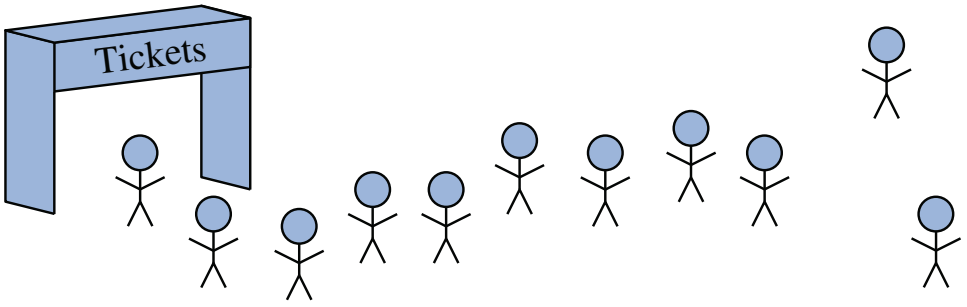
```
1  /** Tests if every opening tag has a matching closing tag in HTML string. */
2  public static boolean isHTMLMatched(String html) {
3    Stack<String> buffer = new LinkedStack<>();
4    int j = html.indexOf('<');                    // find first '<' character (if any)
5    while (j != -1) {
6      int k = html.indexOf('>', j+1);             // find next '>' character
7      if (k == -1)
8        return false;                             // invalid tag
9      String tag = html.substring(j+1, k);        // strip away < >
10     if (!tag.startsWith("/"))                   // this is an opening tag
11       buffer.push(tag);
12     else {                                      // this is a closing tag
13       if (buffer.isEmpty())
14         return false;                           // no tag to match
15       if (!tag.substring(1).equals(buffer.pop()))
16         return false;                           // mismatched tag
17     }
18     j = html.indexOf('<', k+1);                 // find next '<' character (if any)
19   }
20   return buffer.isEmpty();                      // were all opening tags matched?
21 }
```

Code Fragment 6.8: Method for testing if an HTML document has matching tags.

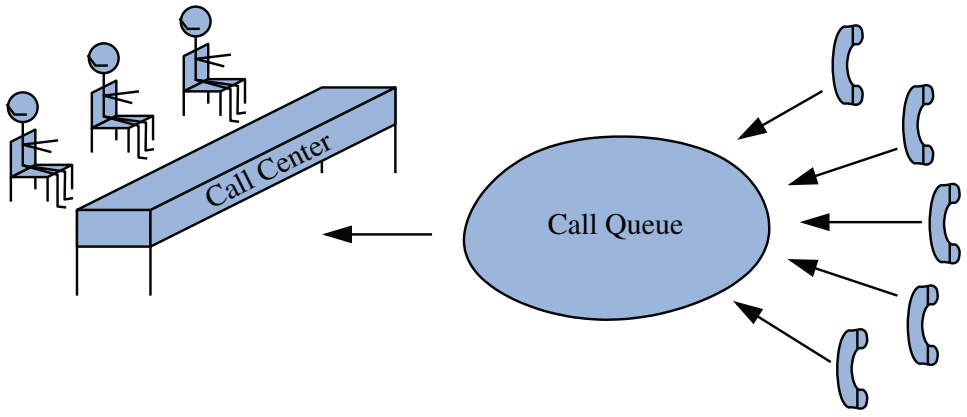
6.2 Queues

Another fundamental data structure is the ***queue***. It is a close "cousin" of the stack, but a queue is a collection of objects that are inserted and removed according to the ***first-in, first-out*** (***FIFO***) principle. That is, elements can be inserted at any time, but only the element that has been in the queue the longest can be next removed.

We usually say that elements enter a queue at the back and are removed from the front. A metaphor for this terminology is a line of people waiting to get on an amusement park ride. People waiting for such a ride enter at the back of the line and get on the ride from the front of the line. There are many other applications of queues (see Figure 6.4). Stores, theaters, reservation centers, and other similar services typically process customer requests according to the FIFO principle. A queue would therefore be a logical choice for a data structure to handle calls to a customer service center, or a wait-list at a restaurant. FIFO queues are also used by many computing devices, such as a networked printer, or a Web server responding to requests.



(a)



(b)

Figure 6.4: Real-world examples of a first-in, first-out queue. (a) People waiting in line to purchase tickets; (b) phone calls being routed to a customer service center.

6.2.1 The Queue Abstract Data Type

Formally, the queue abstract data type defines a collection that keeps objects in a sequence, where element access and deletion are restricted to the ***first*** element in the queue, and element insertion is restricted to the back of the sequence. This restriction enforces the rule that items are inserted and deleted in a queue according to the first-in, first-out (FIFO) principle. The ***queue*** abstract data type (ADT) supports the following two update methods:

enqueue(e): Adds element e to the back of queue.

dequeue( ): Removes and returns the first element from the queue (or null if the queue is empty).

The queue ADT also includes the following accessor methods (with first being analogous to the stack's top method):

first( ): Returns the first element of the queue, without removing it (or null if the queue is empty).

size( ): Returns the number of elements in the queue.

isEmpty( ): Returns a boolean indicating whether the queue is empty.

By convention, we assume that elements added to the queue can have arbitrary type and that a newly created queue is empty. We formalize the queue ADT with the Java interface shown in Code Fragment 6.9.

```
1  public interface Queue<E> {
2    /** Returns the number of elements in the queue. */
3    int size( );
4    /** Tests whether the queue is empty. */
5    boolean isEmpty( );
6    /** Inserts an element at the rear of the queue. */
7    void enqueue(E e);
8    /** Returns, but does not remove, the first element of the queue (null if empty). */
9    E first( );
10   /** Removes and returns the first element of the queue (null if empty). */
11   E dequeue( );
12 }
```

Code Fragment 6.9: A Queue interface defining the queue ADT, with a standard FIFO protocol for insertions and removals.

Example 6.4: *The following table shows a series of queue operations and their effects on an initially empty queue Q of integers.*

Method	Return Value	first $\leftarrow Q \leftarrow$ last
enqueue(5)	–	(5)
enqueue(3)	–	(5, 3)
size( )	2	(5, 3)
dequeue( )	5	(3)
isEmpty( )	false	(3)
dequeue( )	3	( )
isEmpty( )	true	( )
dequeue( )	null	( )
enqueue(7)	–	(7)
enqueue(9)	–	(7, 9)
first( )	7	(7, 9)
enqueue(4)	–	(7, 9, 4)

The java.util.Queue Interface in Java

Java provides a type of queue interface, java.util.Queue, which has functionality similar to the traditional queue ADT, given above, but the documentation for the java.util.Queue interface does not insist that it support only the FIFO principle. When supporting the FIFO principle, the methods of the java.util.Queue interface have the equivalences with the queue ADT shown in Table 6.3.

The java.util.Queue interface supports two styles for most operations, which vary in the way that they treat exceptional cases. When a queue is empty, the remove( ) and element( ) methods throw a NoSuchElementException, while the corresponding methods poll( ) and peek( ) return **null**. For implementations with a bounded capacity, the add method will throw an IllegalStateException when full, while the offer method ignores the new element and returns **false** to signal that the element was not accepted.

Our Queue ADT	Interface java.util.Queue	
	throws exceptions	returns special value
enqueue(e)	add(e)	offer(e)
dequeue( )	remove( )	poll( )
first( )	element( )	peek( )
size( )	size( )	
isEmpty( )	isEmpty( )	

Table 6.3: Methods of the queue ADT and corresponding methods of the interface java.util.Queue, when supporting the FIFO principle.

6.2.2 Array-Based Queue Implementation

In Section 6.1.2, we implemented the LIFO semantics of the Stack ADT using an array (albeit, with a fixed capacity), such that every operation executes in constant time. In this section, we will consider how to use an array to efficiently support the FIFO semantics of the Queue ADT.

Let's assume that as elements are inserted into a queue, we store them in an array such that the first element is at index 0, the second element at index 1, and so on. (See Figure 6.5.)

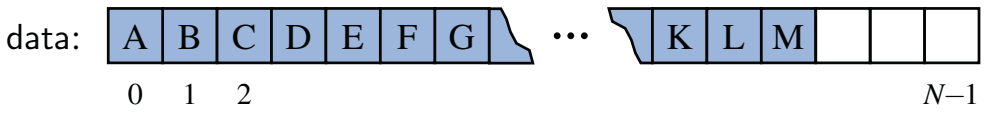


Figure 6.5: Using an array to store elements of a queue, such that the first element inserted, "A", is at cell 0, the second element inserted, "B", at cell 1, and so on.

With such a convention, the question is how we should implement the dequeue operation. The element to be removed is stored at index 0 of the array. One strategy is to execute a loop to shift all other elements of the queue one cell to the left, so that the front of the queue is again aligned with cell 0 of the array. Unfortunately, the use of such a loop would result in an $O(n)$ running time for the dequeue method.

We can improve on the above strategy by avoiding the loop entirely. We will replace a dequeued element in the array with a null reference, and maintain an explicit variable f to represent the index of the element that is currently at the front of the queue. Such an algorithm for dequeue would run in $O(1)$ time. After several dequeue operations, this approach might lead to the configuration portrayed in Figure 6.6.

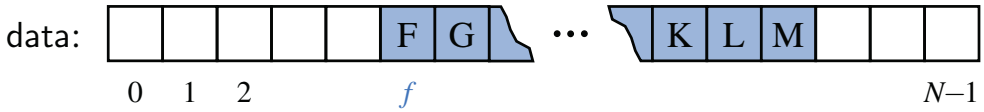


Figure 6.6: Allowing the front of the queue to drift away from index 0. In this representation, index f denotes the location of the front of the queue.

However, there remains a challenge with the revised approach. With an array of capacity N, we should be able to store up to N elements before reaching any exceptional case. If we repeatedly let the front of the queue drift rightward over time, the back of the queue would reach the end of the underlying array even when there are fewer than N elements currently in the queue. We must decide how to store additional elements in such a configuration.

Using an Array Circularly

In developing a robust queue implementation, we allow both the front and back of the queue to drift rightward, with the contents of the queue "wrapping around" the end of an array, as necessary. Assuming that the array has fixed length N, new elements are enqueued toward the "end" of the current queue, progressing from the front to index $N-1$ and continuing at index 0, then 1. Figure 6.7 illustrates such a queue with first element F and last element R.

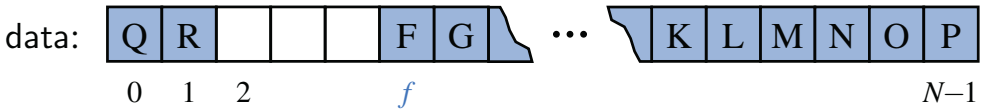


Figure 6.7: Modeling a queue with a circular array that wraps around the end.

Implementing such a circular view is relatively easy with the ***modulo*** operator, denoted with the symbol % in Java. Recall that the modulo operator is computed by taking the remainder after an integral division. For example, 14 divided by 3 has a quotient of 4 with remainder 2, that is, $\frac{14}{3} = 4\frac{2}{3}$. So in Java, 14 / 3 evaluates to the quotient 4, while 14 % 3 evaluates to the remainder 2.

The modulo operator is ideal for treating an array circularly. When we dequeue an element and want to "advance" the front index, we use the arithmetic $f = (f+1) \% N$. As a concrete example, if we have an array of length 10, and a front index 7, we can advance the front by formally computing (7+1) % 10, which is simply 8, as 8 divided by 10 is 0 with a remainder of 8. Similarly, advancing index 8 results in index 9. But when we advance from index 9 (the last one in the array), we compute (9+1) % 10, which evaluates to index 0 (as 10 divided by 10 has a remainder of zero).

A Java Queue Implementation

A complete implementation of a queue ADT using an array in circular fashion is presented in Code Fragment 6.10. Internally, the queue class maintains the following three instance variables:

- data: a reference to the underlying array.
- f: an integer that represents the index, within array data, of the first element of the queue (assuming the queue is not empty).
- sz: an integer representing the current number of elements stored in the queue (not to be confused with the length of the array).

We allow the user to specify the capacity of the queue as an optional parameter to the constructor.

The implementations of methods size and isEmpty are trivial, given the sz field, and the implementation of first is simple, given index f. A discussion of update methods enqueue and dequeue follows the presentation of the code.

```
1  /** Implementation of the queue ADT using a fixed-length array. */
2  public class ArrayQueue<E> implements Queue<E> {
3    // instance variables
4    private E[ ] data;                          // generic array used for storage
5    private int f = 0;                          // index of the front element
6    private int sz = 0;                         // current number of elements
7
8    // constructors
9    public ArrayQueue( ) {this(CAPACITY);} // constructs queue with default capacity
10   public ArrayQueue(int capacity) {          // constructs queue with given capacity
11     data = (E[ ]) new Object[capacity];      // safe cast; compiler may give warning
12   }
13
14   // methods
15   /** Returns the number of elements in the queue. */
16   public int size( ) { return sz; }
17
18   /** Tests whether the queue is empty. */
19   public boolean isEmpty( ) { return (sz == 0); }
20
21   /** Inserts an element at the rear of the queue. */
22   public void enqueue(E e) throws IllegalStateException {
23     if (sz == data.length) throw new IllegalStateException("Queue is full");
24     int avail = (f + sz) % data.length;      // use modular arithmetic
25     data[avail] = e;
26     sz++;
27   }
28
29   /** Returns, but does not remove, the first element of the queue (null if empty). */
30   public E first( ) {
31     if (isEmpty( )) return null;
32     return data[f];
33   }
34
35   /** Removes and returns the first element of the queue (null if empty). */
36   public E dequeue( ) {
37     if (isEmpty( )) return null;
38     E answer = data[f];
39     data[f] = null;                          // dereference to help garbage collection
40     f = (f + 1) % data.length;
41     sz--;
42     return answer;
43   }
```

Code Fragment 6.10: Array-based implementation of a queue.

Adding and Removing Elements

The goal of the enqueue method is to add a new element to the back of the queue. We need to determine the proper index at which to place the new element. Although we do not explicitly maintain an instance variable for the back of the queue, we compute the index of the next opening based on the formula:

```
avail = (f + sz) % data.length;
```

Note that we are using the size of the queue as it exists ***prior*** to the addition of the new element. As a sanity check, for a queue with capacity 10, current size 3, and first element at index 5, its three elements are stored at indices 5, 6, and 7, and the next element should be added at index 8, computed as (5+3) % 10. As a case with wraparound, if the queue has capacity 10, current size 3, and first element at index 8, its three elements are stored at indices 8, 9, and 0, and the next element should be added at index 1, computed as (8+3) % 10.

When the dequeue method is called, the current value of f designates the index of the value that is to be removed and returned. We keep a local reference to the element that will be returned, before setting its cell of the array back to **null**, to aid the garbage collector. Then the index f is updated to reflect the removal of the first element, and the presumed promotion of the second element to become the new first. In most cases, we simply want to increment the index by one, but because of the possibility of a wraparound configuration, we rely on modular arithmetic, computing f = (f+1) % data.length, as originally described on page 242.

Analyzing the Efficiency of an Array-Based Queue

Table 6.4 shows the running times of methods in a realization of a queue by an array. As with our array-based stack implementation, each of the queue methods in the array realization executes a constant number of statements involving arithmetic operations, comparisons, and assignments. Thus, each method in this implementation runs in $O(1)$ time.

Method	Running Time
size	$O(1)$
isEmpty	$O(1)$
first	$O(1)$
enqueue	$O(1)$
dequeue	$O(1)$

Table 6.4: Performance of a queue realized by an array. The space usage is $O(N)$, where N is the size of the array, determined at the time the queue is created, and independent from the number $n < N$ of elements that are actually in the queue.

6.2.3 Implementing a Queue with a Singly Linked List

As we did for the stack ADT, we can easily adapt a singly linked list to implement the queue ADT while supporting worst-case $O(1)$-time for all operations, and without any artificial limit on the capacity. The natural orientation for a queue is to align the front of the queue with the front of the list, and the back of the queue with the tail of the list, because the only update operation that singly linked lists support at the back end is an insertion. Our Java implementation of a LinkedQueue class is given in Code 6.11.

```
1  /** Realization of a FIFO queue as an adaptation of a SinglyLinkedList. */
2  public class LinkedQueue<E> implements Queue<E> {
3    private SinglyLinkedList<E> list = new SinglyLinkedList<>();    // an empty list
4    public LinkedQueue() { }                  // new queue relies on the initially empty list
5    public int size() { return list.size(); }
6    public boolean isEmpty() { return list.isEmpty(); }
7    public void enqueue(E element) { list.addLast(element); }
8    public E first() { return list.first(); }
9    public E dequeue() { return list.removeFirst(); }
10 }
```

Code Fragment 6.11: Implementation of a Queue using a SinglyLinkedList.

Analyzing the Efficiency of a Linked Queue

Although we had not yet introduced asymptotic analysis when we presented our SinglyLinkedList implementation in Chapter 3, it is clear upon reexamination that each method of that class runs in $O(1)$ worst-case time. Therefore, each method of our LinkedQueue adaptation also runs in $O(1)$ worst-case time.

We also avoid the need to specify a maximum size for the queue, as was done in the array-based queue implementation. However, this benefit comes with some expense. Because each node stores a next reference, in addition to the element reference, a linked list uses more space per element than a properly sized array of references.

Also, although all methods execute in constant time for both implementations, it seems clear that the operations involving linked lists have a large number of primitive operations per call. For example, adding an element to an array-based queue consists primarily of calculating an index with modular arithmetic, storing the element in the array cell, and incrementing the size counter. For a linked list, an insertion includes the instantiation and initialization of a new node, relinking an existing node to the new node, and incrementing the size counter. In practice, this makes the linked-list method more expensive than the array-based method.

6.2.4 A Circular Queue

In Section 3.3, we implemented a ***circularly linked list*** class that supports all behaviors of a singly linked list, and an additional rotate( ) method that efficiently moves the first element to the end of the list. We can generalize the Queue interface to define a new CircularQueue interface with such a behavior, as shown in Code Fragment 6.12.

```
1  public interface CircularQueue<E> extends Queue<E> {
2    /**
3     * Rotates the front element of the queue to the back of the queue.
4     * This does nothing if the queue is empty.
5     */
6    void rotate( );
7  }
```

Code Fragment 6.12: A Java interface, CircularQueue, that extends the Queue ADT with a new rotate( ) method.

This interface can easily be implemented by adapting the CircularlyLinkedList class of Section 3.3 to produce a new LinkedCircularQueue class. This class has an advantage over the traditional LinkedQueue, because a call to Q.rotate( ) is implemented more efficiently than the combination of calls, Q.enqueue(Q.dequeue( )), because no nodes are created, destroyed, or relinked by the implementation of a rotate operation on a circularly linked list.

A circular queue is an excellent abstraction for applications in which elements are cyclically arranged, such as for multiplayer, turn-based games, or round-robin scheduling of computing processes. In the remainder of this section, we provide a demonstration of the use of a circular queue.

The Josephus Problem

In the children's game "hot potato," a group of n children sit in a circle passing an object, called the "potato," around the circle. The potato begins with a starting child in the circle, and the children continue passing the potato until a leader rings a bell, at which point the child holding the potato must leave the game after handing the potato to the next child in the circle. After the selected child leaves, the other children close up the circle. This process is then continued until there is only one child remaining, who is declared the winner. If the leader always uses the strategy of ringing the bell so that every k^{th} person is removed from the circle, for some fixed value k, then determining the winner for a given list of children is known as the ***Josephus problem*** (named after an ancient story with far more severe consequences than in the children's game).

Solving the Josephus Problem Using a Queue

We can solve the Josephus problem for a collection of n elements using a circular queue, by associating the potato with the element at the front of the queue and storing elements in the queue according to their order around the circle. Thus, passing the potato is equivalent to rotating the first element to the back of the queue. After this process has been performed $k-1$ times, we remove the front element by dequeuing it from the queue and discarding it. We show a complete Java program for solving the Josephus problem using this approach in Code Fragment 6.13, which describes a solution that runs in $O(nk)$ time. (We can solve this problem faster using techniques beyond the scope of this book.)

```
1  public class Josephus {
2    /** Computes the winner of the Josephus problem using a circular queue. */
3    public static <E> E Josephus(CircularQueue<E> queue, int k) {
4      if (queue.isEmpty( )) return null;
5      while (queue.size( ) > 1) {
6        for (int i=0; i < k−1; i++)    // skip past k-1 elements
7          queue.rotate( );
8        E e = queue.dequeue( );        // remove the front element from the collection
9        System.out.println("       " + e + " is out");
10       }
11     return queue.dequeue( );         // the winner
12   }
13
14   /** Builds a circular queue from an array of objects. */
15   public static <E> CircularQueue<E> buildQueue(E a[ ]) {
16     CircularQueue<E> queue = new LinkedCircularQueue<>( );
17     for (int i=0; i<a.length; i++)
18       queue.enqueue(a[i]);
19     return queue;
20   }
21
22   /** Tester method */
23   public static void main(String[ ] args) {
24     String[ ] a1 = {"Alice", "Bob", "Cindy", "Doug", "Ed", "Fred"};
25     String[ ] a2 = {"Gene", "Hope", "Irene", "Jack", "Kim", "Lance"};
26     String[ ] a3 = {"Mike", "Roberto"};
27     System.out.println("First winner is " + Josephus(buildQueue(a1), 3));
28     System.out.println("Second winner is " + Josephus(buildQueue(a2), 10));
29     System.out.println("Third winner is " + Josephus(buildQueue(a3), 7));
30   }
31 }
```

Code Fragment 6.13: A complete Java program for solving the Josephus problem using a circular queue.

6.3 Double-Ended Queues

We next consider a queue-like data structure that supports insertion and deletion at both the front and the back of the queue. Such a structure is called a ***double-ended queue***, or ***deque***, which is usually pronounced "deck" to avoid confusion with the dequeue method of the regular queue ADT, which is pronounced like the abbreviation "D.Q."

The deque abstract data type is more general than both the stack and the queue ADTs. The extra generality can be useful in some applications. For example, we described a restaurant using a queue to maintain a waitlist. Occasionally, the first person might be removed from the queue only to find that a table was not available; typically, the restaurant will reinsert the person at the *first* position in the queue. It may also be that a customer at the end of the queue may grow impatient and leave the restaurant. (We will need an even more general data structure if we want to model customers leaving the queue from other positions.)

6.3.1 The Deque Abstract Data Type

The deque abstract data type is richer than both the stack and the queue ADTs. To provide a symmetrical abstraction, the deque ADT is defined to support the following update methods:

addFirst(e): Insert a new element e at the front of the deque.

addLast(e): Insert a new element e at the back of the deque.

removeFirst( ): Remove and return the first element of the deque (or null if the deque is empty).

removeLast( ): Remove and return the last element of the deque (or null if the deque is empty).

Additionally, the deque ADT will include the following accessors:

first( ): Returns the first element of the deque, without removing it (or null if the deque is empty).

last( ): Returns the last element of the deque, without removing it (or null if the deque is empty).

size( ): Returns the number of elements in the deque.

isEmpty( ): Returns a boolean indicating whether the deque is empty.

We formalize the deque ADT with the Java interface shown in Code Fragment 6.14.

```
1  /**
2   * Interface for a double-ended queue: a collection of elements that can be inserted
3   * and removed at both ends; this interface is a simplified version of java.util.Deque.
4   */
5  public interface Deque<E> {
6    /** Returns the number of elements in the deque. */
7    int size();
8    /** Tests whether the deque is empty. */
9    boolean isEmpty();
10   /** Returns, but does not remove, the first element of the deque (null if empty). */
11   E first();
12   /** Returns, but does not remove, the last element of the deque (null if empty). */
13   E last();
14   /** Inserts an element at the front of the deque. */
15   void addFirst(E e);
16   /** Inserts an element at the back of the deque. */
17   void addLast(E e);
18   /** Removes and returns the first element of the deque (null if empty). */
19   E removeFirst();
20   /** Removes and returns the last element of the deque (null if empty). */
21   E removeLast();
22 }
```

Code Fragment 6.14: A Java interface, Deque, describing the double-ended queue ADT. Note the use of the generic parameterized type, E, allowing a deque to contain elements of any specified class.

Example 6.5: *The following table shows a series of operations and their effects on an initially empty deque D of integers.*

Method	Return Value	D
addLast(5)	–	(5)
addFirst(3)	–	(3, 5)
addFirst(7)	–	(7, 3, 5)
first()	7	(7, 3, 5)
removeLast()	5	(7, 3)
size()	2	(7, 3)
removeLast()	3	(7)
removeFirst()	7	()
addFirst(6)	–	(6)
last()	6	(6)
addFirst(8)	–	(8, 6)
isEmpty()	false	(8, 6)
last()	6	(8, 6)

6.3.2 Implementing a Deque

We can implement the deque ADT efficiently using either an array or a linked list for storing elements.

Implementing a Deque with a Circular Array

If using an array, we recommend a representation similar to the ArrayQueue class, treating the array in circular fashion and storing the index of the first element and the current size of the deque as fields; the index of the last element can be calculated, as needed, using modular arithmetic.

One extra concern is avoiding use of negative values with the modulo operator. When removing the first element, the front index is advanced in circular fashion, with the assignment f = (f+1) % N. But when an element is inserted at the front, the first index must effectively be decremented in circular fashion and it is a mistake to assign f = (f−1) % N. The problem is that when f is 0, the goal should be to "decrement" it to the other end of the array, and thus to index N−1. However, a calculation such as −1 % 10 in Java results in the value −1. A standard way to decrement an index circularly is instead to assign f = (f−1+N) % N. Adding the additional term of N before the modulus is calculated assures that the result is a positive value. We leave details of this approach to Exercise P-6.40.

Implementing a Deque with a Doubly Linked List

Because the deque requires insertion and removal at both ends, a doubly linked list is most appropriate for implementing all operations efficiently. In fact, the DoublyLinkedList class from Section 3.4.1 already implements the entire Deque interface; we simply need to add the declaration "**implements** Deque<E>" to that class definition in order to use it as a deque.

Performance of the Deque Operations

Table 6.5 shows the running times of methods for a deque implemented with a doubly linked list. Note that every method runs in $O(1)$ time.

Method	Running Time
size, isEmpty	$O(1)$
first, last	$O(1)$
addFirst, addLast	$O(1)$
removeFirst, removeLast	$O(1)$

Table 6.5: Performance of a deque realized by either a circular array or a doubly linked list. The space usage for the array-based implementation is $O(N)$, where N is the size of the array, while the space usage of the doubly linked list is $O(n)$ where $n < N$ is the actual number of elements in the deque.

6.3.3 Deques in the Java Collections Framework

The Java Collections Framework includes its own definition of a deque, as the java.util.Deque interface, as well as several implementations of the interface including one based on use of a circular array (java.util.ArrayDeque) and one based on use of a doubly linked list (java.util.LinkedList). So, if we need to use a deque and would rather not implement one from scratch, we can simply use one of those built-in classes.

As is the case with the java.util.Queue class (see page 240), the java.util.Deque provides duplicative methods that use different techniques to signal exceptional cases. A summary of those methods is given in Table 6.6.

Our Deque ADT	**Interface java.util.Deque**	
	throws exceptions	returns special value
first( )	getFirst( )	peekFirst( )
last( )	getLast( )	peekLast( )
addFirst(e)	addFirst(e)	offerFirst(e)
addLast(e)	addLast(e)	offerLast(e)
removeFirst( )	removeFirst( )	pollFirst( )
removeLast( )	removeLast( )	pollLast( )
size( )	size( )	
isEmpty( )	isEmpty( )	

Table 6.6: Methods of our deque ADT and the corresponding methods of the java.util.Deque interface.

When attempting to access or remove the first or last element of an *empty* deque, the methods in the middle column of Table 6.6—that is, getFirst( ), getLast( ), removeFirst( ), and removeLast( )—throw a NoSuchElementException. The methods in the rightmost column—that is, peekFirst( ), peekLast( ), pollFirst( ), and pollLast( )—simply return the null reference when a deque is empty. In similar manner, when attempting to add an element to an end of a deque with a capacity limit, the addFirst and addLast methods throw an exception, while the offerFirst and offerLast methods return false.

The methods that handle bad situations more gracefully (i.e., without throwing exceptions) are useful in applications, known as producer-consumer scenarios, in which it is common for one component of software to look for an element that may have been placed in a queue by another program, or in which it is common to try to insert an item into a fixed-sized buffer that might be full. However, having methods return null when empty are not appropriate for applications in which null might serve as an actual element of a queue.

6.4 Exercises

Reinforcement

R-6.1 Suppose an initially empty stack S has performed a total of 25 push operations, 12 top operations, and 10 pop operations, 3 of which returned null to indicate an empty stack. What is the current size of S?

R-6.2 Had the stack of the previous problem been an instance of the ArrayStack class, from Code Fragment 6.2, what would be the final value of the instance variable t?

R-6.3 What values are returned during the following series of stack operations, if executed upon an initially empty stack? push(5), push(3), pop(), push(2), push(8), pop(), pop(), push(9), push(1), pop(), push(7), push(6), pop(), pop(), push(4), pop(), pop().

R-6.4 Implement a method with signature transfer(S, T) that transfers all elements from stack S onto stack T, so that the element that starts at the top of S is the first to be inserted onto T, and the element at the bottom of S ends up at the top of T.

R-6.5 Give a recursive method for removing all the elements from a stack.

R-6.6 Give a precise and complete definition of the concept of matching for grouping symbols in an arithmetic expression. Your definition may be recursive.

R-6.7 Suppose an initially empty queue Q has performed a total of 32 enqueue operations, 10 first operations, and 15 dequeue operations, 5 of which returned null to indicate an empty queue. What is the current size of Q?

R-6.8 Had the queue of the previous problem been an instance of the ArrayQueue class, from Code Fragment 6.10, with capacity 30 never exceeded, what would be the final value of the instance variable f?

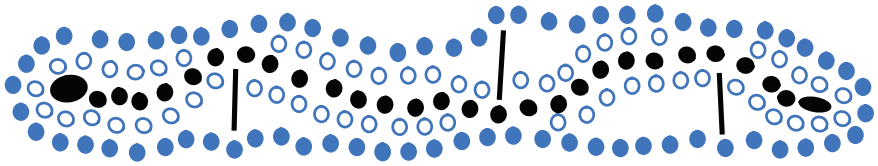
R-6.9 What values are returned during the following sequence of queue operations, if executed on an initially empty queue? enqueue(5), enqueue(3), dequeue(), enqueue(2), enqueue(8), dequeue(), dequeue(), enqueue(9), enqueue(1), dequeue(), enqueue(7), enqueue(6), dequeue(), dequeue(), enqueue(4), dequeue(), dequeue().

R-6.10 Give a simple adapter that implements the stack ADT while using an instance of a deque for storage.

R-6.11 Give a simple adapter that implements the queue ADT while using an instance of a deque for storage.

R-6.12 What values are returned during the following sequence of deque ADT operations, on an initially empty deque? addFirst(3), addLast(8), addLast(9), addFirst(1), last(), isEmpty(), addFirst(2), removeLast(), addLast(7), first(), last(), addLast(4), size(), removeFirst(), removeFirst().

Chapter 7 List and Iterator ADTs



Contents

7.1 The List ADT

In Chapter 6, we introduced the stack, queue, and deque abstract data types, and discussed how either an array or a linked list could be used for storage in an efficient concrete implementation of each. Each of those ADTs represents a linearly ordered sequence of elements. The deque is the most general of the three, yet even so, it only allows insertions and deletions at the front or back of a sequence.

In this chapter, we explore several abstract data types that represent a linear sequence of elements, but with more general support for adding or removing elements at arbitrary positions. However, designing a single abstraction that is well suited for efficient implementation with either an array or a linked list is challenging, given the very different nature of these two fundamental data structures.

Locations within an array are easily described with an integer ***index***. Recall that an index of an element e in a sequence is equal to the number of elements before e in that sequence. By this definition, the first element of a sequence has index 0, and the last has index $n-1$, assuming that n denotes the total number of elements. The notion of an element's index is well defined for a linked list as well, although we will see that it is not as convenient of a notion, as there is no way to efficiently access an element at a given index without traversing a portion of the linked list that depends upon the magnitude of the index.

With that said, Java defines a general interface, java.util.List, that includes the following index-based methods (and more):

size( ): Returns the number of elements in the list.

isEmpty( ): Returns a boolean indicating whether the list is empty.

get(i): Returns the element of the list having index i; an error condition occurs if i is not in range $[0, \text{size}( ) - 1]$.

set(i, e): Replaces the element at index i with e, and returns the old element that was replaced; an error condition occurs if i is not in range $[0, \text{size}( ) - 1]$.

add(i, e): Inserts a new element e into the list so that it has index i, moving all subsequent elements one index later in the list; an error condition occurs if i is not in range $[0, \text{size}( )]$.

remove(i): Removes and returns the element at index i, moving all subsequent elements one index earlier in the list; an error condition occurs if i is not in range $[0, \text{size}( ) - 1]$.

We note that the index of an existing element may change over time, as other elements are added or removed in front of it. We also draw attention to the fact that the range of valid indices for the add method includes the current size of the list, in which case the new element becomes the last.

Example 7.1 demonstrates a series of operations on a list instance, and Code Fragment 7.1 below provides a formal definition of our simplified version of the List interface; we use an IndexOutOfBoundsException to signal an invalid index argument.

Example 7.1: *We demonstrate operations on an initially empty list of characters.*

Method	Return Value	List Contents
add(0, A)	–	(A)
add(0, B)	–	(B, A)
get(1)	A	(B, A)
set(2, C)	"error"	(B, A)
add(2, C)	–	(B, A, C)
add(4, D)	"error"	(B, A, C)
remove(1)	A	(B, C)
add(1, D)	–	(B, D, C)
add(1, E)	–	(B, E, D, C)
get(4)	"error"	(B, E, D, C)
add(4, F)	–	(B, E, D, C, F)
set(2, G)	D	(B, E, G, C, F)
get(2)	G	(B, E, G, C, F)

```
1  /** A simplified version of the java.util.List interface. */
2  public interface List<E> {
3    /** Returns the number of elements in this list. */
4    int size( );
5
6    /** Returns whether the list is empty. */
7    boolean isEmpty( );
8
9    /** Returns (but does not remove) the element at index i. */
10   E get(int i) throws IndexOutOfBoundsException;
11
12   /** Replaces the element at index i with e, and returns the replaced element. */
13   E set(int i, E e) throws IndexOutOfBoundsException;
14
15   /** Inserts element e to be at index i, shifting all subsequent elements later. */
16   void add(int i, E e) throws IndexOutOfBoundsException;
17
18   /** Removes/returns the element at index i, shifting subsequent elements earlier. */
19   E remove(int i) throws IndexOutOfBoundsException;
20 }
```

Code Fragment 7.1: A simple version of the List interface.

7.2 Array Lists

An obvious choice for implementing the list ADT is to use an array A, where $A[i]$ stores (a reference to) the element with index i. We will begin by assuming that we have a fixed-capacity array, but in Section 7.2.1 describe a more advanced technique that effectively allows an array-based list to have unbounded capacity. Such an unbounded list is known as an ***array list*** in Java (or a ***vector*** in C++ and in the earliest versions of Java).

With a representation based on an array A, the get(i) and set(i, e) methods are easy to implement by accessing $A[i]$ (assuming i is a legitimate index). Methods add(i, e) and remove(i) are more time consuming, as they require shifting elements up or down to maintain our rule of always storing an element whose list index is i at index i of the array. (See Figure 7.1.) Our initial implementation of the ArrayList class follows in Code Fragments 7.2 and 7.3.

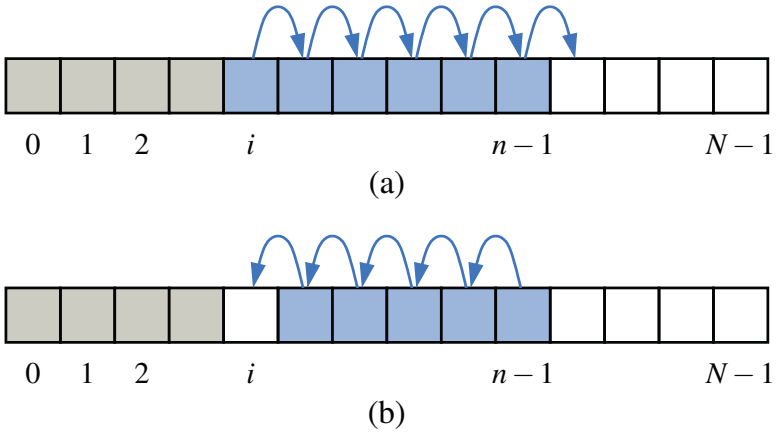


Figure 7.1: Array-based implementation of an array list that is storing n elements: (a) shifting up for an insertion at index i; (b) shifting down for a removal at index i.

```
1  public class ArrayList<E> implements List<E> {
2    // instance variables
3    public static final int CAPACITY=16;     // default array capacity
4    private E[ ] data;                       // generic array used for storage
5    private int size = 0;                    // current number of elements
6    // constructors
7    public ArrayList( ) { this(CAPACITY); }  // constructs list with default capacity
8    public ArrayList(int capacity) {         // constructs list with given capacity
9      data = (E[ ]) new Object[capacity];    // safe cast; compiler may give warning
10   }
```

Code Fragment 7.2: An implementation of a simple ArrayList class with bounded capacity. (Continues in Code Fragment 7.3.)

```
11    // public methods
12    /** Returns the number of elements in the array list. */
13    public int size( ) { return size; }
14    /** Returns whether the array list is empty. */
15    public boolean isEmpty( ) { return size == 0; }
16    /** Returns (but does not remove) the element at index i. */
17    public E get(int i) throws IndexOutOfBoundsException {
18      checkIndex(i, size);
19      return data[i];
20    }
21    /** Replaces the element at index i with e, and returns the replaced element. */
22    public E set(int i, E e) throws IndexOutOfBoundsException {
23      checkIndex(i, size);
24      E temp = data[i];
25      data[i] = e;
26      return temp;
27    }
28    /** Inserts element e to be at index i, shifting all subsequent elements later. */
29    public void add(int i, E e) throws IndexOutOfBoundsException,
30                                                IllegalStateException {
31      checkIndex(i, size + 1);
32      if (size == data.length)                  // not enough capacity
33        throw new IllegalStateException("Array is full");
34      for (int k=size-1; k >= i; k--)           // start by shifting rightmost
35        data[k+1] = data[k];
36      data[i] = e;                              // ready to place the new element
37      size++;
38    }
39    /** Removes/returns the element at index i, shifting subsequent elements earlier. */
40    public E remove(int i) throws IndexOutOfBoundsException {
41      checkIndex(i, size);
42      E temp = data[i];
43      for (int k=i; k < size-1; k++)            // shift elements to fill hole
44        data[k] = data[k+1];
45      data[size-1] = null;                      // help garbage collection
46      size--;
47      return temp;
48    }
49    // utility method
50    /** Checks whether the given index is in the range [0, n-1]. */
51    protected void checkIndex(int i, int n) throws IndexOutOfBoundsException {
52      if (i < 0 || i >= n)
53        throw new IndexOutOfBoundsException("Illegal index: " + i);
54    }
55  }
```

Code Fragment 7.3: An implementation of a simple ArrayList class with bounded capacity. (Continued from Code Fragment 7.2.)

The Performance of a Simple Array-Based Implementation

Table 7.1 shows the worst-case running times of the methods of an array list with n elements realized by means of an array. Methods isEmpty, size, get and set clearly run in $O(1)$ time, but the insertion and removal methods can take much longer than this. In particular, add(i, e) runs in time $O(n)$. Indeed, the worst case for this operation occurs when i is 0, since all the existing n elements have to be shifted forward. A similar argument applies to method remove(i), which runs in $O(n)$ time, because we have to shift backward $n-1$ elements in the worst case, when i is 0. In fact, assuming that each possible index is equally likely to be passed as an argument to these operations, their average running time is $O(n)$, for we will have to shift $n/2$ elements on average.

Method	Running Time
size( )	$O(1)$
isEmpty( )	$O(1)$
get(i)	$O(1)$
set(i, e)	$O(1)$
add(i, e)	$O(n)$
remove(i)	$O(n)$

Table 7.1: Performance of an array list with n elements realized by a fixed-capacity array.

Looking more closely at add(i, e) and remove(i), we note that they each run in time $O(n-i+1)$, for only those elements at index i and higher have to be shifted up or down. Thus, inserting or removing an item at the end of an array list, using the methods add(n, e) and remove$(n-1)$ respectively, takes $O(1)$ time each. Moreover, this observation has an interesting consequence for the adaptation of the array list ADT to the deque ADT from Section 6.3.1. If we do the "obvious" thing and store elements of a deque so that the first element is at index 0 and the last element at index $n-1$, then methods addLast and removeLast of the deque each run in $O(1)$ time. However, methods addFirst and removeFirst of the deque each run in $O(n)$ time.

Actually, with a little effort, we can produce an array-based implementation of the array list ADT that achieves $O(1)$ time for insertions and removals at index 0, as well as insertions and removals at the end of the array list. Achieving this requires that we give up on our rule that an element at index i is stored in the array at index i, however, as we would have to use a circular array approach like the one we used in Section 6.2 to implement a queue. We leave the details of this implementation as Exercise C-7.25.

7.2.1 Dynamic Arrays

The ArrayList implementation in Code Fragments 7.2 and 7.3 (as well as those for a stack, queue, and deque from Chapter 6) has a serious limitation; it requires that a fixed maximum capacity be declared, throwing an exception if attempting to add an element once full. This is a major weakness, because if a user is unsure of the maximum size that will be reached for a collection, there is risk that either too large of an array will be requested, causing an inefficient waste of memory, or that too small of an array will be requested, causing a fatal error when exhausting that capacity.

Java's ArrayList class provides a more robust abstraction, allowing a user to add elements to the list, with no apparent limit on the overall capacity. To provide this abstraction, Java relies on an algorithmic sleight of hand that is known as a ***dynamic array***.

In reality, elements of an ArrayList are stored in a traditional array, and the precise size of that traditional array must be internally declared in order for the system to properly allocate a consecutive piece of memory for its storage. For example, Figure 7.2 displays an array with 12 cells that might be stored in memory locations 2146 through 2157 on a computer system.

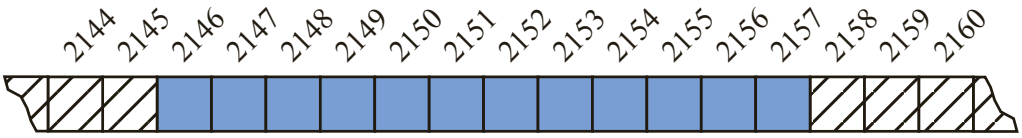


Figure 7.2: An array of 12 cells, allocated in memory locations 2146 through 2157.

Because the system may allocate neighboring memory locations to store other data, the capacity of an array cannot be increased by expanding into subsequent cells.

The first key to providing the semantics of an unbounded array is that an array list instance maintains an internal array that often has greater capacity than the current length of the list. For example, while a user may have created a list with five elements, the system may have reserved an underlying array capable of storing eight object references (rather than only five). This extra capacity makes it easy to add a new element to the end of the list by using the next available cell of the array.

If a user continues to add elements to a list, all reserved capacity in the underlying array will eventually be exhausted. In that case, the class requests a new, larger array from the system, and copies all references from the smaller array into the beginning of the new array. At that point in time, the old array is no longer needed, so it can be reclaimed by the system. Intuitively, this strategy is much like that of the hermit crab, which moves into a larger shell when it outgrows its previous one.

7.2.2 Implementing a Dynamic Array

We now demonstrate how our original version of the ArrayList, from Code Fragments 7.2 and 7.3, can be transformed to a dynamic-array implementation, having unbounded capacity. We rely on the same internal representation, with a traditional array A, that is initialized either to a default capacity or to one specified as a parameter to the constructor.

The key is to provide means to "grow" the array A, when more space is needed. Of course, we cannot actually grow that array, as its capacity is fixed. Instead, when a call to add a new element risks ***overflowing*** the current array, we perform the following additional steps:

1. Allocate a new array B with larger capacity.
2. Set $B[k] = A[k]$, for $k = 0, \ldots, n-1$, where n denotes current number of items.
3. Set $A = B$, that is, we henceforth use the new array to support the list.
4. Insert the new element in the new array.

An illustration of this process is shown in Figure 7.3.

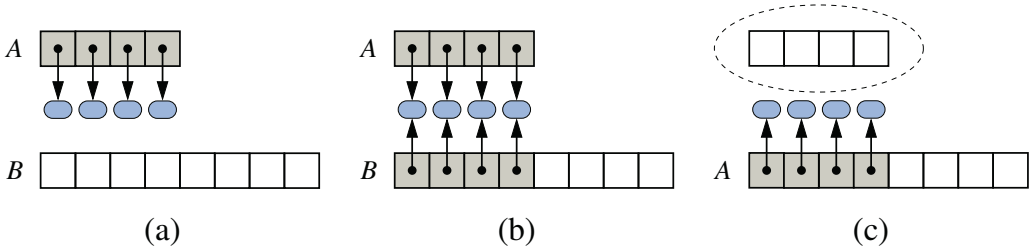


Figure 7.3: An illustration of "growing" a dynamic array: (a) create new array B; (b) store elements of A in B; (c) reassign reference A to the new array. Not shown is the future garbage collection of the old array, or the insertion of a new element.

Code Fragment 7.4 provides a concrete implementation of a resize method, which should be included as a protected method within the original ArrayList class. The instance variable data corresponds to array A in the above discussion, and local variable temp corresponds to array B.

```
/** Resizes internal array to have given capacity >= size. */
protected void resize(int capacity) {
  E[ ] temp = (E[ ]) new Object[capacity]; // safe cast; compiler may give warning
  for (int k=0; k < size; k++)
    temp[k] = data[k];
  data = temp;                            // start using the new array
}
```

Code Fragment 7.4: An implementation of the ArrayList.resize method.

The remaining issue to consider is how large of a new array to create. A commonly used rule is for the new array to have twice the capacity of the existing array that has been filled. In Section 7.2.3, we will provide a mathematical analysis to justify such a choice.

To complete the revision to our original ArrayList implementation, we redesign the add method so that it calls the new resize utility when detecting that the current array is filled (rather than throwing an exception). The revised version appears in Code Fragment 7.5.

```
28   /** Inserts element e to be at index i, shifting all subsequent elements later. */
29   public void add(int i, E e) throws IndexOutOfBoundsException {
30     checkIndex(i, size + 1);
31     if (size == data.length)                    // not enough capacity
32       resize(2 * data.length);                  // so double the current capacity
...    // rest of method unchanged...
```

Code Fragment 7.5: A revision to the ArrayList.add method, originally from Code Fragment 7.3, which calls the resize method of Code Fragment 7.4 when more capacity is needed.

Finally, we note that our original implementation of the ArrayList class includes two constructors: a default constructor that uses an initial capacity of 16, and a parameterized constructor that allows the caller to specify a capacity value. With the use of dynamic arrays, that capacity is no longer a fixed limit. Still, greater efficiency is achieved when a user selects an initial capacity that matches the actual size of a data set, as this can avoid time spent on intermediate array reallocations and potential space that is wasted by having too large of an array.

7.2.3 Amortized Analysis of Dynamic Arrays

In this section, we will perform a detailed analysis of the running time of operations on dynamic arrays. As a shorthand notation, let us refer to the insertion of an element to be the last element in an array list as a ***push*** operation.

The strategy of replacing an array with a new, larger array might at first seem slow, because a single push operation may require $\Omega(n)$ time to perform, where n is the current number of elements in the array. (Recall, from Section 4.3.1, that big-Omega notation, describes an asymptotic lower bound on the running time of an algorithm.) However, by doubling the capacity during an array replacement, our new array allows us to add n further elements before the array must be replaced again. In this way, there are many simple push operations for each expensive one (see Figure 7.4). This fact allows us to show that a series of push operations on an initially empty dynamic array is efficient in terms of its total running time.

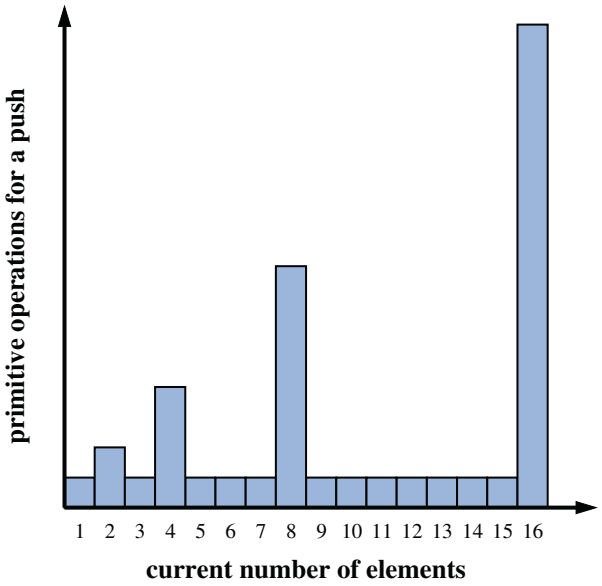


Figure 7.4: Running times of a series of push operations on a dynamic array.

Using an algorithmic design pattern called ***amortization***, we show that performing a sequence of push operations on a dynamic array is actually quite efficient. To perform an ***amortized analysis***, we use an accounting technique where we view the computer as a coin-operated appliance that requires the payment of one ***cyber-dollar*** for a constant amount of computing time. When an operation is executed, we should have enough cyber-dollars available in our current "bank account" to pay for that operation's running time. Thus, the total amount of cyber-dollars spent for any computation will be proportional to the total time spent on that computation. The beauty of using this analysis method is that we can overcharge some operations in order to save up cyber-dollars to pay for others.

Proposition 7.2: *Let L be an initially empty array list with capacity one, implemented by means of a dynamic array that doubles in size when full. The total time to perform a series of n push operations in L is $O(n)$.*

Justification: Let us assume that one cyber-dollar is enough to pay for the execution of each push operation in L, excluding the time spent for growing the array. Also, let us assume that growing the array from size k to size $2k$ requires k cyber-dollars for the time spent initializing the new array. We shall charge each push operation three cyber-dollars. Thus, we overcharge each push operation that does not cause an overflow by two cyber-dollars. Think of the two cyber-dollars profited in an insertion that does not grow the array as being "stored" with the cell in which the element was inserted. An overflow occurs when the array L has 2^i elements, for some integer $i \geq 0$, and the size of the array used by the array representing L is 2^i. Thus, doubling the size of the array will require 2^i cyber-dollars. Fortunately, these cyber-dollars can be found stored in cells 2^{i-1} through $2^i - 1$. (See Figure 7.5.)

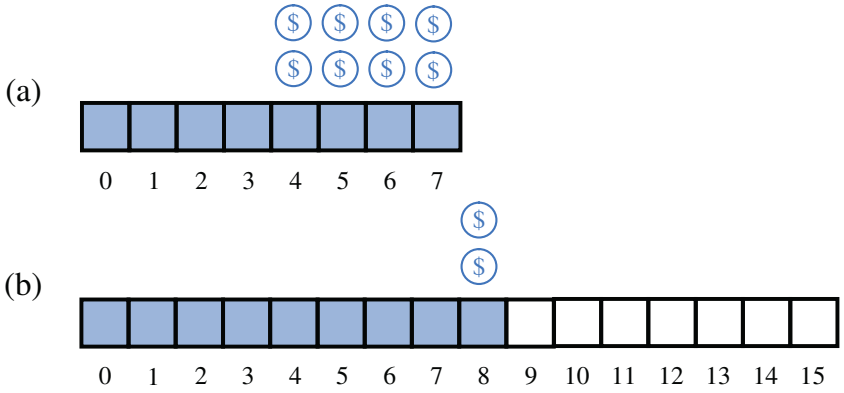


Figure 7.5: Illustration of a series of push operations on a dynamic array: (a) an 8-cell array is full, with two cyber-dollars "stored" at cells 4 through 7; (b) a push operation causes an overflow and a doubling of capacity. Copying the eight old elements to the new array is paid for by the cyber-dollars already stored in the table. Inserting the new element is paid for by one of the cyber-dollars charged to the current push operation, and the two cyber-dollars profited are stored at cell 8.

Note that the previous overflow occurred when the number of elements became larger than 2^{i-1} for the first time, and thus the cyber-dollars stored in cells 2^{i-1} through $2^i - 1$ have not yet been spent. Therefore, we have a valid amortization scheme in which each operation is charged three cyber-dollars and all the computing time is paid for. That is, we can pay for the execution of n push operations using $3n$ cyber-dollars. In other words, the amortized running time of each push operation is $O(1)$; hence, the total running time of n push operations is $O(n)$. ■

Geometric Increase in Capacity

Although the proof of Proposition 7.2 relies on the array being doubled each time it is expanded, the $O(1)$ amortized bound per operation can be proven for any geometrically increasing progression of array sizes. (See Section 2.2.3 for discussion of geometric progressions.) When choosing the geometric base, there exists a trade-off between runtime efficiency and memory usage. If the last insertion causes a resize event, with a base of 2 (i.e., doubling the array), the array essentially ends up twice as large as it needs to be. If we instead increase the array by only 25% of its current size (i.e., a geometric base of 1.25), we do not risk wasting as much memory in the end, but there will be more intermediate resize events along the way. Still it is possible to prove an $O(1)$ amortized bound, using a constant factor greater than the 3 cyber-dollars per operation used in the proof of Proposition 7.2 (see Exercise R-7.7). The key to the performance is that the amount of additional space is proportional to the current size of the array.

Beware of Arithmetic Progression

To avoid reserving too much space at once, it might be tempting to implement a dynamic array with a strategy in which a constant number of additional cells are reserved each time an array is resized. Unfortunately, the overall performance of such a strategy is significantly worse. At an extreme, an increase of only one cell causes each push operation to resize the array, leading to a familiar $1+2+3+\cdots+n$ summation and $\Omega(n^2)$ overall cost. Using increases of 2 or 3 at a time is slightly better, as portrayed in Figure 7.4, but the overall cost remains quadratic.

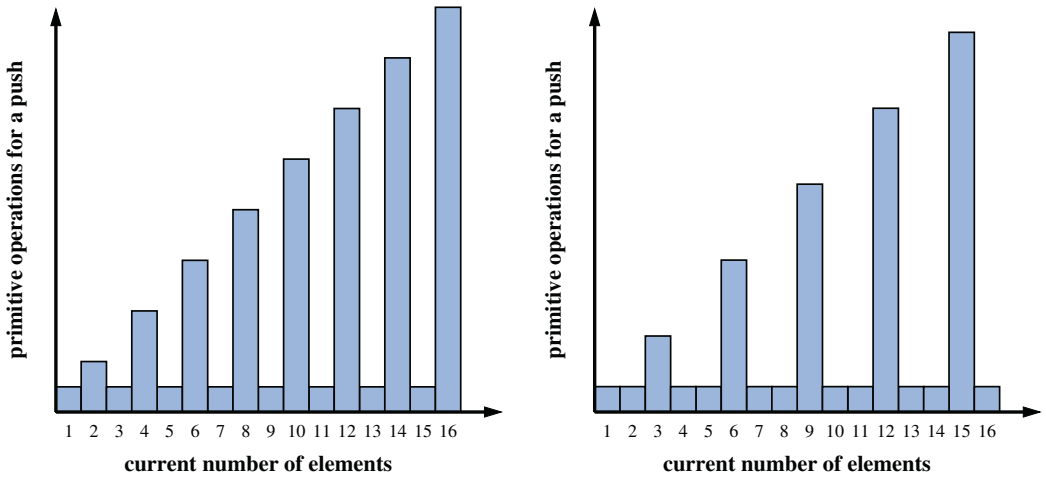


Figure 7.6: Running times of a series of push operations on a dynamic array using arithmetic progression of sizes. Part (a) assumes an increase of 2 in the size of the array, while part (b) assumes an increase of 3.

Using a *fixed* increment for each resize, and thus an arithmetic progression of intermediate array sizes, results in an overall time that is quadratic in the number of operations, as shown in the following proposition. In essence, even an increase in 10,000 cells per resize will become insignificant for large data sets.

Proposition 7.3: *Performing a series of n push operations on an initially empty dynamic array using a fixed increment with each resize takes $\Omega(n^2)$ time.*

Justification: Let $c > 0$ represent the fixed increment in capacity that is used for each resize event. During the series of n push operations, time will have been spent initializing arrays of size $c, 2c, 3c, \ldots, mc$ for $m = \lceil n/c \rceil$, and therefore, the overall time is proportional to $c+2c+3c+\cdots+mc$. By Proposition 4.3, this sum is

$$\sum_{i=1}^{m} ci = c \cdot \sum_{i=1}^{m} i = c\frac{m(m+1)}{2} \geq c\frac{\frac{n}{c}(\frac{n}{c}+1)}{2} \geq \frac{1}{2c} \cdot n^2.$$

Therefore, performing the n push operations takes $\Omega(n^2)$ time. ■

Memory Usage and Shrinking an Array

Another consequence of the rule of a geometric increase in capacity when adding to a dynamic array is that the final array size is guaranteed to be proportional to the overall number of elements. That is, the data structure uses $O(n)$ memory. This is a very desirable property for a data structure.

If a container, such as an array list, provides operations that cause the removal of one or more elements, greater care must be taken to ensure that a dynamic array guarantees $O(n)$ memory usage. The risk is that repeated insertions may cause the underlying array to grow arbitrarily large, and that there will no longer be a proportional relationship between the actual number of elements and the array capacity after many elements are removed.

A robust implementation of such a data structure will shrink the underlying array, on occasion, while maintaining the $O(1)$ amortized bound on individual operations. However, care must be taken to ensure that the structure cannot rapidly oscillate between growing and shrinking the underlying array, in which case the amortized bound would not be achieved. In Exercise C-7.29, we explore a strategy in which the array capacity is halved whenever the number of actual element falls below one-fourth of that capacity, thereby guaranteeing that the array capacity is at most four times the number of elements; we explore the amortized analysis of such a strategy in Exercises C-7.30 and C-7.31.

7.2.4 Java's StringBuilder class

Near the beginning of Chapter 4, we described an experiment in which we compared two algorithms for composing a long string (Code Fragment 4.2). The first of those relied on repeated concatenation using the String class, and the second relied on use of Java's StringBuilder class. We observed the StringBuilder was significantly faster, with empirical evidence that suggested a quadratic running time for the algorithm with repeated concatenations, and a linear running time for the algorithm with the StringBuilder. We are now able to explain the theoretical underpinning for those observations.

The StringBuilder class represents a mutable string by storing characters in a dynamic array. With analysis similar to Proposition 7.2, it guarantees that a series of append operations resulting in a string of length n execute in a combined time of $O(n)$. (Insertions at positions other than the end of a string builder do not carry this guarantee, just as they do not for an ArrayList.)

In contrast, the repeated use of string concatenation requires quadratic time. We originally analyzed that algorithm on page 172 of Chapter 4. In effect, that approach is akin to a dynamic array with an arithmetic progression of size one, repeatedly copying all characters from one array to a new array with size one greater than before.

7.3 Positional Lists

When working with array-based sequences, integer indices provide an excellent means for describing the location of an element, or the location at which an insertion or deletion should take place. However, numeric indices are not a good choice for describing positions within a linked list because, knowing only an element's index, the only way to reach it is to traverse the list incrementally from its beginning or end, counting elements along the way.

Furthermore, indices are not a good abstraction for describing a more local view of a position in a sequence, because the index of an entry changes over time due to insertions or deletions that happen earlier in the sequence. For example, it may not be convenient to describe the location of a person waiting in line based on the index, as that requires knowledge of precisely how far away that person is from the front of the line. We prefer an abstraction, as characterized in Figure 7.7, in which there is some other means for describing a position.

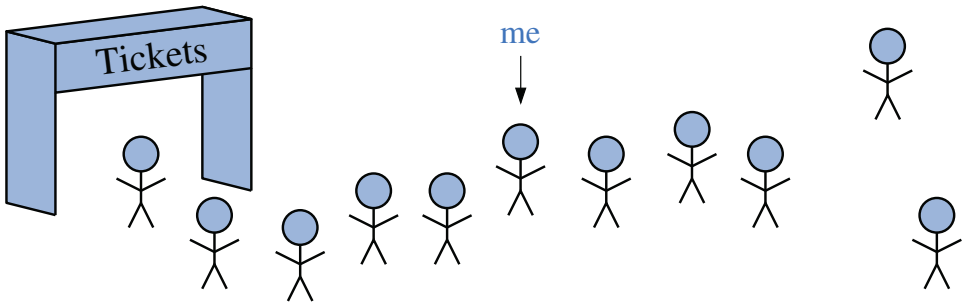


Figure 7.7: We wish to be able to identify the position of an element in a sequence without the use of an integer index. The label "me" represents some abstraction that identifies the position.

Our goal is to design an abstract data type that provides a user a way to refer to elements anywhere in a sequence, and to perform arbitrary insertions and deletions. This would allow us to efficiently describe actions such as a person deciding to leave the line before reaching the front, or allowing a friend to "cut" into line right behind him or her.

As another example, a text document can be viewed as a long sequence of characters. A word processor uses the abstraction of a ***cursor*** to describe a position within the document without explicit use of an integer index, allowing operations such as "delete the character at the cursor" or "insert a new character just after the cursor." Furthermore, we may be able to refer to an inherent position within a document, such as the beginning of a particular chapter, without relying on a character index (or even a chapter number) that may change as the document evolves.

For these reasons, we temporarily forego the index-based methods of Java's formal List interface, and instead develop our own abstract data type that we denote as a ***positional list***. Although a positional list is an abstraction, and need not rely on a linked list for its implementation, we certainly have a linked list in mind as we design the ADT, ensuring that it takes best advantage of particular capabilities of a linked list, such as $O(1)$-time insertions and deletions at arbitrary positions (something that is not possible with an array-based sequence).

We face an immediate challenge in designing the ADT; to achieve constant time insertions and deletions at arbitrary locations, we effectively need a reference to the node at which an element is stored. It is therefore very tempting to develop an ADT in which a node reference serves as the mechanism for describing a position. In fact, our DoublyLinkedList class of Section 3.4.1 has methods addBetween and remove that accept node references as parameters; however, we intentionally declared those methods as private.

Unfortunately, the public use of nodes in the ADT would violate the object-oriented design principles of abstraction and encapsulation, which were introduced in Chapter 2. There are several reasons to prefer that we encapsulate the nodes of a linked list, for both our sake and for the benefit of users of our abstraction:

- It will be simpler for users of our data structure if they are not bothered with unnecessary details of our implementation, such as low-level manipulation of nodes, or our reliance on the use of sentinel nodes. Notice that to use the addBetween method of our DoublyLinkedList class to add a node at the beginning of a sequence, the header sentinel must be sent as a parameter.
- We can provide a more robust data structure if we do not permit users to directly access or manipulate the nodes. We can then ensure that users do not invalidate the consistency of a list by mismanaging the linking of nodes. A more subtle problem arises if a user were allowed to call the addBetween or remove method of our DoublyLinkedList class, sending a node that does not belong to the given list as a parameter. (Go back and look at that code and see why it causes a problem!)
- By better encapsulating the internal details of our implementation, we have greater flexibility to redesign the data structure and improve its performance. In fact, with a well-designed abstraction, we can provide a notion of a nonnumeric position, even if using an array-based sequence. (See Exercise C-7.43.)

Therefore, in defining the positional list ADT, we also introduce the concept of a ***position***, which formalizes the intuitive notion of the "location" of an element relative to others in the list. (When we do use a linked list for the implementation, we will later see how we can privately use node references as natural manifestations of positions.)

7.3.1 Positions

To provide a general abstraction for the location of an element within a structure, we define a simple ***position*** abstract data type. A position supports the following single method:

getElement( ): Returns the element stored at this position.

A position acts as a marker or token within a broader positional list. A position p, which is associated with some element e in a list L, does not change, even if the index of e changes in L due to insertions or deletions elsewhere in the list. Nor does position p change if we replace the element e stored at p with another element. The only way in which a position becomes invalid is if that position (and its element) are explicitly removed from the list.

Having a formal definition of a position type allows positions to serve as parameters to some methods and return values from other methods of the positional list ADT, which we next describe.

7.3.2 The Positional List Abstract Data Type

We now view a ***positional list*** as a collection of positions, each of which stores an element. The accessor methods provided by the positional list ADT include the following, for a list L:

first( ): Returns the position of the first element of L (or null if empty).

last( ): Returns the position of the last element of L (or null if empty).

before(p): Returns the position of L immediately before position p (or null if p is the first position).

after(p): Returns the position of L immediately after position p (or null if p is the last position).

isEmpty( ): Returns true if list L does not contain any elements.

size( ): Returns the number of elements in list L.

An error occurs if a position p, sent as a parameter to a method, is not a valid position for the list.

Note well that the first( ) and last( ) methods of the positional list ADT return the associated *positions*, not the *elements*. (This is in contrast to the corresponding first and last methods of the deque ADT.) The first element of a positional list can be determined by subsequently invoking the getElement method on that position, as first( ).getElement. The advantage of receiving a position as a return value is that we can subsequently use that position to traverse the list.

As a demonstration of a typical traversal of a positional list, Code Fragment 7.6 traverses a list, named guests, that stores string elements, and prints each element while traversing from the beginning of the list to the end.

```
1 Position<String> cursor = guests.first( );
2 while (cursor != null) {
3   System.out.println(cursor.getElement( ));
4   cursor = guests.after(cursor);              // advance to the next position (if any)
5 }
```

Code Fragment 7.6: A traversal of a positional list.

This code relies on the convention that the null reference is returned when the after method is called upon the last position. (That return value is clearly distinguishable from any legitimate position.) The positional list ADT similarly indicates that the null value is returned when the before method is invoked at the front of the list, or when first or last methods are called upon an empty list. Therefore, the above code fragment works correctly even if the guests list is empty.

Updated Methods of a Positional List

The positional list ADT also includes the following ***update*** methods:

addFirst(e): Inserts a new element e at the front of the list, returning the position of the new element.

addLast(e): Inserts a new element e at the back of the list, returning the position of the new element.

addBefore(p, e): Inserts a new element e in the list, just before position p, returning the position of the new element.

addAfter(p, e): Inserts a new element e in the list, just after position p, returning the position of the new element.

set(p, e): Replaces the element at position p with element e, returning the element formerly at position p.

remove(p): Removes and returns the element at position p in the list, invalidating the position.

There may at first seem to be redundancy in the above repertoire of operations for the positional list ADT, since we can perform operation addFirst(e) with addBefore(first( ), e), and operation addLast(e) with addAfter(last( ), e). But these substitutions can only be done for a nonempty list.

Example 7.4: *The following table shows a series of operations on an initially empty positional list storing integers. To identify position instances, we use variables such as p and q. For ease of exposition, when displaying the list contents, we use subscript notation to denote the position storing an element.*

Method	Return Value	List Contents
addLast(8)	p	(8_p)
first( )	p	(8_p)
addAfter(p, 5)	q	$(8_p, 5_q)$
before(q)	p	$(8_p, 5_q)$
addBefore(q, 3)	r	$(8_p, 3_r, 5_q)$
r.getElement( )	3	$(8_p, 3_r, 5_q)$
after(p)	r	$(8_p, 3_r, 5_q)$
before(p)	null	$(8_p, 3_r, 5_q)$
addFirst(9)	s	$(9_s, 8_p, 3_r, 5_q)$
remove(last( ))	5	$(9_s, 8_p, 3_r)$
set(p, 7)	8	$(9_s, 7_p, 3_r)$
remove(q)	"error"	$(9_s, 7_p, 3_r)$

Java Interface Definitions

We are now ready to formalize the position ADT and positional list ADT. A Java Position interface, representing the position ADT, is given in Code Fragment 7.7. Following that, Code Fragment 7.8 presents a Java definition for our PositionalList interface. If the getElement( ) method is called on a Position instance that has previously been removed from its list, an IllegalStateException is thrown. If an invalid Position instance is sent as a parameter to a method of a PositionalList, an IllegalArgumentException is thrown. (Both of those exception types are defined in the standard Java hierarchy.)

```
1  public interface Position<E> {
2    /**
3     * Returns the element stored at this position.
4     *
5     * @return the stored element
6     * @throws IllegalStateException if position no longer valid
7     */
8    E getElement( ) throws IllegalStateException;
9  }
```

Code Fragment 7.7: The Position interface.

```
1  /** An interface for positional lists. */
2  public interface PositionalList<E> {
3
4    /** Returns the number of elements in the list. */
5    int size( );
6
7    /** Tests whether the list is empty. */
8    boolean isEmpty( );
9
10   /** Returns the first Position in the list (or null, if empty). */
11   Position<E> first( );
12
13   /** Returns the last Position in the list (or null, if empty). */
14   Position<E> last( );
15
16   /** Returns the Position immediately before Position p (or null, if p is first). */
17   Position<E> before(Position<E> p) throws IllegalArgumentException;
18
19   /** Returns the Position immediately after Position p (or null, if p is last). */
20   Position<E> after(Position<E> p) throws IllegalArgumentException;
21
22   /** Inserts element e at the front of the list and returns its new Position. */
23   Position<E> addFirst(E e);
24
25   /** Inserts element e at the back of the list and returns its new Position. */
26   Position<E> addLast(E e);
27
28   /** Inserts element e immediately before Position p and returns its new Position. */
29   Position<E> addBefore(Position<E> p, E e)
30     throws IllegalArgumentException;
31
32   /** Inserts element e immediately after Position p and returns its new Position. */
33   Position<E> addAfter(Position<E> p, E e)
34     throws IllegalArgumentException;
35
36   /** Replaces the element stored at Position p and returns the replaced element. */
37   E set(Position<E> p, E e) throws IllegalArgumentException;
38
39   /** Removes the element stored at Position p and returns it (invalidating p). */
40   E remove(Position<E> p) throws IllegalArgumentException;
41 }
```

Code Fragment 7.8: The PositionalList interface.

7.3.3 Doubly Linked List Implementation

Not surprisingly, our preferred implementation of the PositionalList interface relies on a doubly linked list. Although we implemented a DoublyLinkedList class in Chapter 3, that class does not adhere to the PositionalList interface.

In this section, we develop a concrete implementation of the PositionalList interface using a doubly linked list. The low-level details of our new linked-list representation, such as the use of header and trailer sentinels, will be identical to our earlier version; we refer the reader to Section 3.4 for a discussion of the doubly linked list operations. What differs in this section is our management of the positional abstraction.

The obvious way to identify locations within a linked list are node references. Therefore, we declare the nested Node class of our linked list so as to implement the Position interface, supporting the required getElement method. So the nodes *are* the positions. Yet, the Node class is declared as private, to maintain proper encapsulation. All of the public methods of the positional list rely on the Position type, so although we know we are sending and receiving nodes, these are only known to be positions from the outside; as a result, users of our class cannot call any method other than getElement( ).

In Code Fragments 7.9–7.12, we define a LinkedPositionalList class, which implements the positional list ADT. We provide the following guide to that code:

- Code Fragment 7.9 contains the definition of the nested Node<E> class, which implements the Position<E> interface. Following that are the declaration of the instance variables of the outer LinkedPositionalList class and its constructor.
- Code Fragment 7.10 begins with two important utility methods that help us robustly cast between the Position and Node types. The validate(p) method is called anytime the user sends a Position instance as a parameter. It throws an exception if it determines that the position is invalid, and otherwise returns that instance, implicitly cast as a Node, so that methods of the Node class can subsequently be called. The private position(node) method is used when about to return a Position to the user. Its primary purpose is to make sure that we do not expose either sentinel node to a caller, returning a **null** reference in such a case. We rely on both of these private utility methods in the public accessor methods that follow.
- Code Fragment 7.11 provides most of the public update methods, relying on a private addBetween method to unify the implementations of the various insertion operations.
- Code Fragment 7.12 provides the public remove method. Note that it sets all fields of the removed node back to null—a condition we can later detect to recognize a defunct position.

```
1  /** Implementation of a positional list stored as a doubly linked list. */
2  public class LinkedPositionalList<E> implements PositionalList<E> {
3    //---------- nested Node class ----------
4    private static class Node<E> implements Position<E> {
5      private E element;                  // reference to the element stored at this node
6      private Node<E> prev;               // reference to the previous node in the list
7      private Node<E> next;               // reference to the subsequent node in the list
8      public Node(E e, Node<E> p, Node<E> n) {
9        element = e;
10       prev = p;
11       next = n;
12     }
13     public E getElement( ) throws IllegalStateException {
14       if (next == null)                          // convention for defunct node
15         throw new IllegalStateException("Position no longer valid");
16       return element;
17     }
18     public Node<E> getPrev( ) {
19       return prev;
20     }
21     public Node<E> getNext( ) {
22       return next;
23     }
24     public void setElement(E e) {
25       element = e;
26     }
27     public void setPrev(Node<E> p) {
28       prev = p;
29     }
30     public void setNext(Node<E> n) {
31       next = n;
32     }
33   } //---------- end of nested Node class ----------
34
35   // instance variables of the LinkedPositionalList
36   private Node<E> header;                       // header sentinel
37   private Node<E> trailer;                      // trailer sentinel
38   private int size = 0;                         // number of elements in the list
39
40   /** Constructs a new empty list. */
41   public LinkedPositionalList( ) {
42     header = new Node<>(null, null, null);      // create header
43     trailer = new Node<>(null, header, null);   // trailer is preceded by header
44     header.setNext(trailer);                    // header is followed by trailer
45   }
```

Code Fragment 7.9: An implementation of the LinkedPositionalList class. (Continues in Code Fragments 7.10–7.12.)

```
46    // private utilities
47    /** Validates the position and returns it as a node. */
48    private Node<E> validate(Position<E> p) throws IllegalArgumentException {
49      if (!(p instanceof Node)) throw new IllegalArgumentException("Invalid p");
50      Node<E> node = (Node<E>) p;     // safe cast
51      if (node.getNext( ) == null)       // convention for defunct node
52        throw new IllegalArgumentException("p is no longer in the list");
53      return node;
54    }
55
56    /** Returns the given node as a Position (or null, if it is a sentinel). */
57    private Position<E> position(Node<E> node) {
58      if (node == header || node == trailer)
59        return null;     // do not expose user to the sentinels
60      return node;
61    }
62
63    // public accessor methods
64    /** Returns the number of elements in the linked list. */
65    public int size( ) { return size; }
66
67    /** Tests whether the linked list is empty. */
68    public boolean isEmpty( ) { return size == 0; }
69
70    /** Returns the first Position in the linked list (or null, if empty). */
71    public Position<E> first( ) {
72      return position(header.getNext( ));
73    }
74
75    /** Returns the last Position in the linked list (or null, if empty). */
76    public Position<E> last( ) {
77      return position(trailer.getPrev( ));
78    }
79
80    /** Returns the Position immediately before Position p (or null, if p is first). */
81    public Position<E> before(Position<E> p) throws IllegalArgumentException {
82      Node<E> node = validate(p);
83      return position(node.getPrev( ));
84    }
85
86    /** Returns the Position immediately after Position p (or null, if p is last). */
87    public Position<E> after(Position<E> p) throws IllegalArgumentException {
88      Node<E> node = validate(p);
89      return position(node.getNext( ));
90    }
```

Code Fragment 7.10: An implementation of the LinkedPositionalList class. (Continued from Code Fragment 7.9; continues in Code Fragments 7.11 and 7.12.)

```
91    // private utilities
92    /** Adds element e to the linked list between the given nodes. */
93    private Position<E> addBetween(E e, Node<E> pred, Node<E> succ) {
94      Node<E> newest = new Node<>(e, pred, succ);  // create and link a new node
95      pred.setNext(newest);
96      succ.setPrev(newest);
97      size++;
98      return newest;
99    }
100
101   // public update methods
102   /** Inserts element e at the front of the linked list and returns its new Position. */
103   public Position<E> addFirst(E e) {
104     return addBetween(e, header, header.getNext( ));          // just after the header
105   }
106
107   /** Inserts element e at the back of the linked list and returns its new Position. */
108   public Position<E> addLast(E e) {
109     return addBetween(e, trailer.getPrev( ), trailer);          // just before the trailer
110   }
111
112   /** Inserts element e immediately before Position p, and returns its new Position.*/
113   public Position<E> addBefore(Position<E> p, E e)
114                                       throws IllegalArgumentException {
115     Node<E> node = validate(p);
116     return addBetween(e, node.getPrev( ), node);
117   }
118
119   /** Inserts element e immediately after Position p, and returns its new Position. */
120   public Position<E> addAfter(Position<E> p, E e)
121                                       throws IllegalArgumentException {
122     Node<E> node = validate(p);
123     return addBetween(e, node, node.getNext( ));
124   }
125
126   /** Replaces the element stored at Position p and returns the replaced element. */
127   public E set(Position<E> p, E e) throws IllegalArgumentException {
128     Node<E> node = validate(p);
129     E answer = node.getElement( );
130     node.setElement(e);
131     return answer;
132   }
```

Code Fragment 7.11: An implementation of the LinkedPositionalList class. (Continued from Code Fragments 7.9 and 7.10; continues in Code Fragment 7.12.)

```
133   /** Removes the element stored at Position p and returns it (invalidating p). */
134   public E remove(Position<E> p) throws IllegalArgumentException {
135     Node<E> node = validate(p);
136     Node<E> predecessor = node.getPrev( );
137     Node<E> successor = node.getNext( );
138     predecessor.setNext(successor);
139     successor.setPrev(predecessor);
140     size−−;
141     E answer = node.getElement( );
142     node.setElement(null);                    // help with garbage collection
143     node.setNext(null);                       // and convention for defunct node
144     node.setPrev(null);
145     return answer;
146   }
147 }
```

Code Fragment 7.12: An implementation of the LinkedPositionalList class. (Continued from Code Fragments 7.9–7.11.)

The Performance of a Linked Positional List

The positional list ADT is ideally suited for implementation with a doubly linked list, as all operations run in worst-case constant time, as shown in Table 7.2. This is in stark contrast to the ArrayList structure (analyzed in Table 7.1), which requires linear time for insertions or deletions at arbitrary positions, due to the need for a loop to shift other elements.

Of course, our positional list does not support the index-based methods of the official List interface of Section 7.1. It is possible to add support for those methods by traversing the list while counting nodes (see Exercise C-7.38), but that requires time proportional to the sublist that is traversed.

Method	Running Time
size( )	$O(1)$
isEmpty( )	$O(1)$
first( ), last( )	$O(1)$
before(p), after(p)	$O(1)$
addFirst(e), addLast(e)	$O(1)$
addBefore(p, e), addAfter(p, e)	$O(1)$
set(p, e)	$O(1)$
remove(p)	$O(1)$

Table 7.2: Performance of a positional list with n elements realized by a doubly linked list. The space usage is $O(n)$.

Implementing a Positional List with an Array

We can implement a positional list L using an array A for storage, but some care is necessary in designing objects that will serve as positions. At first glance, it would seem that a position p need only store the index i at which its associated element is stored within the array. We can then implement method getElement(p) simply by returning $A[i]$. The problem with this approach is that the index of an element e changes when other insertions or deletions occur before it. If we have already returned a position p associated with element e that stores an outdated index i to a user, the wrong array cell would be accessed when the position was used. (Remember that positions in a positional list should always be defined relative to their neighboring positions, not their indices.)

Hence, if we are going to implement a positional list with an array, we need a different approach. We recommend the following representation. Instead of storing the elements of L directly in array A, we store a new kind of position object in each cell of A. A position p stores the element e as well as the current index i of that element within the list. Such a data structure is illustrated in Figure 7.8.

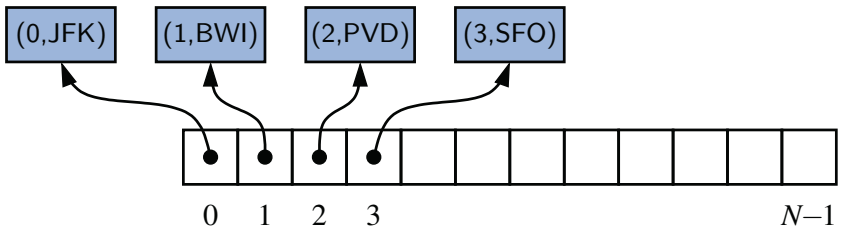


Figure 7.8: An array-based representation of a positional list.

With this representation, we can determine the index currently associated with a position, and we can determine the position currently associated with a specific index. We can therefore implement an accessor, such as before(p), by finding the index of the given position and using the array to find the neighboring position.

When an element is inserted or deleted somewhere in the list, we can loop through the array to update the index variable stored in all later positions in the list that are shifted during the update.

Efficiency Trade-Offs with an Array-Based Sequence

In this array implementation of a sequence, the addFirst, addBefore, addAfter, and remove methods take $O(n)$ time, because we have to shift position objects to make room for the new position or to fill in the hole created by the removal of the old position (just as in the insert and remove methods based on index). All the other position-based methods take $O(1)$ time.

7.4 Iterators

An ***iterator*** is a software design pattern that abstracts the process of scanning through a sequence of elements, one element at a time. The underlying elements might be stored in a container class, streaming through a network, or generated by a series of computations.

In order to unify the treatment and syntax for iterating objects in a way that is independent from a specific organization, Java defines the java.util.Iterator interface with the following two methods:

hasNext( ): Returns true if there is at least one additional element in the sequence, and false otherwise.

next( ): Returns the next element in the sequence.

The interface uses Java's generic framework, with the next( ) method returning a parameterized element type. For example, the Scanner class (described in Section 1.6) formally implements the Iterator<String> interface, with its next( ) method returning a String instance.

If the next( ) method of an iterator is called when no further elements are available, a NoSuchElementException is thrown. Of course, the hasNext( ) method can be used to detect that condition before calling next( ).

The combination of these two methods allows a general loop construct for processing elements of the iterator. For example, if we let variable, iter, denote an instance of the Iterator<String> type, then we can write the following:

```
while (iter.hasNext( )) {
  String value = iter.next( );
  System.out.println(value);
}
```

The java.util.Iterator interface contains a third method, which is *optionally* supported by some iterators:

remove( ): Removes from the collection the element returned by the most recent call to next( ). Throws an IllegalStateException if next has not yet been called, or if remove was already called since the most recent call to next.

This method can be used to filter a collection of elements, for example to discard all negative numbers from a data set.

For the sake of simplicity, we will not implement the remove method for most data structures in this book, but we will give two tangible examples later in this section. If removal is not supported, an UnsupportedOperationException is conventionally thrown.

7.4.1 The Iterable Interface and Java's For-Each Loop

A single iterator instance supports only one pass through a collection; calls to next can be made until all elements have been reported, but there is no way to "reset" the iterator back to the beginning of the sequence.

However, a data structure that wishes to allow repeated iterations can support a method that returns a *new* iterator, each time it is called. To provide greater standardization, Java defines another parameterized interface, named Iterable, that includes the following single method:

iterator( ): Returns an iterator of the elements in the collection.

An instance of a typical collection class in Java, such as an ArrayList, is ***iterable*** (but not itself an ***iterator***); it produces an iterator for its collection as the return value of the iterator( ) method. Each call to iterator( ) returns a new iterator instance, thereby allowing multiple (even simultaneous) traversals of a collection.

Java's Iterable class also plays a fundamental role in support of the "for-each" loop syntax (described in Section 1.5.2). The loop syntax,

```
for (ElementType variable : collection) {
  loopBody                                    // may refer to "variable"
}
```

is supported for any instance, *collection*, of an iterable class. *ElementType* must be the type of object returned by its iterator, and *variable* will take on element values within the *loopBody*. Essentially, this syntax is shorthand for the following:

```
Iterator<ElementType> iter = collection.iterator( );
while (iter.hasNext( )) {
  ElementType variable = iter.next( );
  loopBody                                    // may refer to "variable"
}
```

We note that the iterator's remove method cannot be invoked when using the for-each loop syntax. Instead, we must explicitly use an iterator. As an example, the following loop can be used to remove all negative numbers from an ArrayList of floating-point values.

```
ArrayList<Double> data;  // populate with random numbers (not shown)
Iterator<Double> walk = data.iterator( );
while (walk.hasNext( ))
  if (walk.next( ) < 0.0)
    walk.remove( );
```

7.4.2 Implementing Iterators

There are two general styles for implementing iterators that differ in terms of what work is done when the iterator instance is first created, and what work is done each time the iterator is advanced with a call to next().

A ***snapshot iterator*** maintains its own private copy of the sequence of elements, which is constructed at the time the iterator object is created. It effectively records a "snapshot" of the sequence of elements at the time the iterator is created, and is therefore unaffected by any subsequent changes to the primary collection that may occur. Implementing snapshot iterators tends to be very easy, as it requires a simple traversal of the primary structure. The downside of this style of iterator is that it requires $O(n)$ time and $O(n)$ auxiliary space, upon construction, to copy and store a collection of n elements.

A ***lazy iterator*** is one that does not make an upfront copy, instead performing a piecewise traversal of the primary structure only when the next() method is called to request another element. The advantage of this style of iterator is that it can typically be implemented so the iterator requires only $O(1)$ space and $O(1)$ construction time. One downside (or feature) of a lazy iterator is that its behavior is affected if the primary structure is modified (by means other than by the iterator's own remove method) before the iteration completes. Many of the iterators in Java's libraries implement a "fail-fast" behavior that immediately invalidates such an iterator if its underlying collection is modified unexpectedly.

We will demonstrate how to implement iterators for both the ArrayList and LinkedPositionalList classes as examples. We implement lazy iterators for both, including support for the remove operation (but without any fail-fast guarantee).

Iterations with the ArrayList class

We begin by discussing iteration for the ArrayList<E> class. We will have it implement the Iterable<E> interface. (In fact, that requirement is already part of Java's List interface.) Therefore, we must add an iterator() method to that class definition, which returns an instance of an object that implements the Iterator<E> interface. For this purpose, we define a new class, ArrayIterator, as a nonstatic nested class of ArrayList (i.e., an ***inner class***, as described in Section 2.6). The advantage of having the iterator as an inner class is that it can access private fields (such as the array A) that are members of the containing list.

Our implementation is given in Code Fragment 7.13. The iterator() method of ArrayList returns a new instance of the inner ArrayIterator class. Each iterator maintains a field j that represents the index of the next element to be returned. It is initialized to 0, and when j reaches the size of the list, there are no more elements to return. In order to support element removal through the iterator, we also maintain a boolean variable that denotes whether a call to remove is currently permissible.

```
1   //---------- nested ArrayIterator class ----------
2   /**
3    * A (nonstatic) inner class. Note well that each instance contains an implicit
4    * reference to the containing list, allowing it to access the list's members.
5    */
6   private class ArrayIterator implements Iterator<E> {
7     private int j = 0;                        // index of the next element to report
8     private boolean removable = false;        // can remove be called at this time?
9
10    /**
11     * Tests whether the iterator has a next object.
12     * @return true if there are further objects, false otherwise
13     */
14    public boolean hasNext( ) { return j < size; }  // size is field of outer instance
15
16    /**
17     * Returns the next object in the iterator.
18     *
19     * @return next object
20     * @throws NoSuchElementException if there are no further elements
21     */
22    public E next( ) throws NoSuchElementException {
23      if (j == size) throw new NoSuchElementException("No next element");
24      removable = true;     // this element can subsequently be removed
25      return data[j++];     // post-increment j, so it is ready for future call to next
26    }
27
28    /**
29     * Removes the element returned by most recent call to next.
30     * @throws IllegalStateException if next has not yet been called
31     * @throws IllegalStateException if remove was already called since recent next
32     */
33    public void remove( ) throws IllegalStateException {
34      if (!removable) throw new IllegalStateException("nothing to remove");
35      ArrayList.this.remove(j-1);     // that was the last one returned
36      j--;                            // next element has shifted one cell to the left
37      removable = false;              // do not allow remove again until next is called
38    }
39  } //---------- end of nested ArrayIterator class ----------
40
41  /** Returns an iterator of the elements stored in the list. */
42  public Iterator<E> iterator( ) {
43    return new ArrayIterator( );        // create a new instance of the inner class
44  }
```

Code Fragment 7.13: Code providing support for ArrayList iterators. (This should be nested within the ArrayList class definition of Code Fragments 7.2 and 7.3.)

Iterations with the LinkedPositionalList class

In support the concept of iteration with the LinkedPositionalList class, a first question is whether to support iteration of the *elements* of the list or the *positions* of the list. If we allow a user to iterate through all positions of the list, those positions could be used to access the underlying elements, so support for position iteration is more general. However, it is more standard for a container class to support iteration of the core elements, by default, so that the for-each loop syntax could be used to write code such as the following,

```
for (String guest : waitlist)
```

assuming that variable waitlist has type LinkedPositionalList<String>.

For maximum convenience, we will support *both* forms of iteration. We will have the standard iterator( ) method return an iterator of the elements of the list, so that our list class formally implements the Iterable interface for the declared element type.

For those wishing to iterate through the positions of a list, we will provide a new method, positions( ). At first glance, it would seem a natural choice for such a method to return an Iterator. However, we prefer for the return type of that method to be an instance that is Iterable (and hence, has its own iterator( ) method that returns an iterator of positions). Our reason for the extra layer of complexity is that we wish for users of our class to be able to use a for-each loop with a simple syntax such as the following:

```
for (Position<String> p : waitlist.positions( ))
```

For this syntax to be legal, the return type of positions( ) must be Iterable.

Code Fragment 7.14 presents our new support for the iteration of positions and elements of a LinkedPositionalList. We define three new inner classes. The first of these is PositionIterator, providing the core functionality of our list iterations. Whereas the array list iterator maintained the index of the next element to be returned as a field, this class maintains the position of the next element to be returned (as well as the position of the most recently returned element, to support removal).

To support our goal of the positions( ) method returning an Iterable object, we define a trivial PositionIterable inner class, which simply constructs and returns a new PositionIterator object each time its iterator( ) method is called. The positions( ) method of the top-level class returns a new PositionIterable instance. Our framework relies heavily on these being inner classes, not static nested classes.

Finally, we wish to have the top-level iterator( ) method return an iterator of elements (not positions). Rather than reinvent the wheel, we trivially adapt the PositionIterator class to define a new ElementIterator class, which lazily manages a position iterator instance, while returning the element stored at each position when next( ) is called.

```
1   //---------- nested PositionIterator class ----------
2   private class PositionIterator implements Iterator<Position<E>> {
3     private Position<E> cursor = first( );    // position of the next element to report
4     private Position<E> recent = null;        // position of last reported element
5     /** Tests whether the iterator has a next object. */
6     public boolean hasNext( ) { return (cursor != null);    }
7     /** Returns the next position in the iterator. */
8     public Position<E> next( ) throws NoSuchElementException {
9       if (cursor == null) throw new NoSuchElementException("nothing left");
10      recent = cursor;                // element at this position might later be removed
11      cursor = after(cursor);
12      return recent;
13    }
14    /** Removes the element returned by most recent call to next. */
15    public void remove( ) throws IllegalStateException {
16      if (recent == null) throw new IllegalStateException("nothing to remove");
17      LinkedPositionalList.this.remove(recent);          // remove from outer list
18      recent = null;                  // do not allow remove again until next is called
19    }
20  } //---------- end of nested PositionIterator class ----------
21
22  //---------- nested PositionIterable class ----------
23  private class PositionIterable implements Iterable<Position<E>> {
24    public Iterator<Position<E>> iterator( ) { return new PositionIterator( ); }
25  } //---------- end of nested PositionIterable class ----------
26
27  /** Returns an iterable representation of the list's positions. */
28  public Iterable<Position<E>> positions( ) {
29    return new PositionIterable( );          // create a new instance of the inner class
30  }
31
32  //---------- nested ElementIterator class ----------
33  /* This class adapts the iteration produced by positions() to return elements. */
34  private class ElementIterator implements Iterator<E> {
35    Iterator<Position<E>> posIterator = new PositionIterator( );
36    public boolean hasNext( ) { return posIterator.hasNext( ); }
37    public E next( ) { return posIterator.next( ).getElement( ); } // return element!
38    public void remove( ) { posIterator.remove( ); }
39  }
40
41  /** Returns an iterator of the elements stored in the list. */
42  public Iterator<E> iterator( ) { return new ElementIterator( ); }
```

Code Fragment 7.14: Support for providing iterations of positions and elements of a LinkedPositionalList. (This should be nested within the LinkedPositionalList class definition of Code Fragments 7.9–7.12.)

7.5 The Java Collections Framework

Java provides many data structure interfaces and classes, which together form the ***Java Collections Framework***. This framework, which is part of the java.util package, includes versions of several of the data structures discussed in this book, some of which we have already discussed and others of which we will discuss later in this book. The root interface in the Java collections framework is named Collection. This is a general interface for any data structure, such as a list, that represents a collection of elements. The Collection interface includes many methods, including some we have already seen (e.g., size( ), isEmpty( ), iterator( )). It is a superinterface for other interfaces in the Java Collections Framework that can hold elements, including the java.util interfaces Deque, List, and Queue, and other subinterfaces discussed later in this book, including Set (Section 10.5.1) and Map (Section 10.1).

The Java Collections Framework also includes concrete classes implementing various interfaces with a combination of properties and underlying representations. We summarize but a few of those classes in Table 7.3. For each, we denote which of the Queue, Deque, or List interfaces are implemented (possibly several). We also discuss several behavioral properties. Some classes enforce, or allow, a fixed capacity limit. Robust classes provide support for ***concurrency***, allowing multiple processes to share use of a data structure in a thread-safe manner. If the structure is designated as ***blocking***, a call to retrieve an element from an empty collection waits until some other process inserts an element. Similarly, a call to insert into a full blocking structure must wait until room becomes available.

	Interfaces			**Properties**			**Storage**	
Class	Queue	Deque	List	Capacity Limit	Thread-Safe	Blocking	Array	Linked List
ArrayBlockingQueue	✓			✓	✓	✓	✓	
LinkedBlockingQueue	✓			✓	✓	✓		✓
ConcurrentLinkedQueue	✓				✓		✓	
ArrayDeque	✓	✓					✓	
LinkedBlockingDeque	✓	✓		✓	✓	✓		✓
ConcurrentLinkedDeque	✓	✓			✓			✓
ArrayList			✓				✓	
LinkedList	✓	✓	✓					✓

Table 7.3: Several classes in the Java Collections Framework.

7.5.1 List Iterators in Java

The java.util.LinkedList class does not expose a position concept to users in its API, as we do in our positional list ADT. Instead, the preferred way to access and update a LinkedList object in Java, without using indices, is to use a ListIterator that is returned by the list's listIterator( ) method. Such an iterator provides forward and backward traversal methods as well as local update methods. It views its current position as being before the first element, between two elements, or after the last element. That is, it uses a list ***cursor***, much like a screen cursor is viewed as being located between two characters on a screen. Specifically, the java.util.ListIterator interface includes the following methods:

add(e): Adds the element e at the current position of the iterator.

hasNext( ): Returns true if there is an element after the current position of the iterator.

hasPrevious( ): Returns true if there is an element before the current position of the iterator.

previous( ): Returns the element e before the current position and sets the current position to be before e.

next( ): Returns the element e after the current position and sets the current position to be after e.

nextIndex( ): Returns the index of the next element.

previousIndex( ): Returns the index of the previous element.

remove( ): Removes the element returned by the most recent next or previous operation.

set(e): Replaces the element returned by the most recent call to the next or previous operation with e.

It is risky to use multiple iterators over the same list while modifying its contents. If insertions, deletions, or replacements are required at multiple "places" in a list, it is safer to use positions to specify these locations. But the java.util.LinkedList class does not expose its position objects to the user. So, to avoid the risks of modifying a list that has created multiple iterators, the iterators have a "fail-fast" feature that invalidates such an iterator if its underlying collection is modified unexpectedly. For example, if a java.util.LinkedList object L has returned five different iterators and one of them modifies L, a ConcurrentModificationException is thrown if any of the other four is subsequently used. That is, Java allows many list iterators to be traversing a linked list L at the same time, but if one of them modifies L (using an add, set, or remove method), then all the other iterators for L become invalid. Likewise, if L is modified by one of its own update methods, then all existing iterators for L immediately become invalid.

7.5.2 Comparison to Our Positional List ADT

Java provides functionality similar to our array list and positional lists ADT in the java.util.List interface, which is implemented with an array in java.util.ArrayList and with a linked list in java.util.LinkedList.

Moreover, Java uses iterators to achieve a functionality similar to what our positional list ADT derives from positions. Table 7.4 shows corresponding methods between our (array and positional) list ADTs and the java.util interfaces List and ListIterator interfaces, with notes about their implementations in the java.util classes ArrayList and LinkedList.

Positional List ADT Method	**java.util.List Method**	**ListIterator Method**	**Notes**
size( )	size( )		$O(1)$ time
isEmpty( )	isEmpty( )		$O(1)$ time
	get(i)		A is $O(1)$, L is $O(\min\{i, n-i\})$
first( )	listIterator( )		first element is next
last( )	listIterator(size( ))		last element is previous
before(p)		previous( )	$O(1)$ time
after(p)		next( )	$O(1)$ time
set(p, e)		set(e)	$O(1)$ time
	set(i, e)		A is $O(1)$, L is $O(\min\{i, n-i\})$
	add(i, e)		$O(n)$ time
addFirst(e)	add(0, e)		A is $O(n)$, L is $O(1)$
addFirst(e)	addFirst(e)		only exists in L, $O(1)$
addLast(e)	add(e)		$O(1)$ time
addLast(e)	addLast(e)		only exists in L, $O(1)$
addAfter(p, e)		add(e)	insertion is at cursor; A is $O(n)$, L is $O(1)$
addBefore(p, e)		add(e)	insertion is at cursor; A is $O(n)$, L is $O(1)$
remove(p)		remove( )	deletion is at cursor; A is $O(n)$, L is $O(1)$
	remove(i)		A is $O(1)$, L is $O(\min\{i, n-i\})$

Table 7.4: Correspondences between methods in our positional list ADT and the java.util interfaces List and ListIterator. We use A and L as abbreviations for java.util.ArrayList and java.util.LinkedList (or their running times).

7.5.3 List-Based Algorithms in the Java Collections Framework

In addition to the classes that are provided in the Java Collections Framework, there are a number of simple algorithms that it provides as well. These algorithms are implemented as static methods in the java.util.Collections class (not to be confused with the java.util.Collection interface) and they include the following methods:

copy(L_{dest}, L_{src}): Copies all elements of the L_{src} list into corresponding indices of the L_{dest} list.

disjoint(C, D): Returns a boolean value indicating whether the collections C and D are disjoint.

fill(L, e): Replaces each element of the list L with element e.

frequency(C, e): Returns the number of elements in the collection C that are equal to e.

max(C): Returns the maximum element in the collection C, based on the natural ordering of its elements.

min(C): Returns the minimum element in the collection C, based on the natural ordering of its elements.

replaceAll(L, e, f): Replaces each element in L that is equal to e with element f.

reverse(L): Reverses the ordering of elements in the list L.

rotate(L, d): Rotates the elements in the list L by the distance d (which can be negative), in a circular fashion.

shuffle(L): Pseudorandomly permutes the ordering of the elements in the list L.

sort(L): Sorts the list L, using the natural ordering of its elements.

swap(L, i, j): Swap the elements at indices i and j of list L.