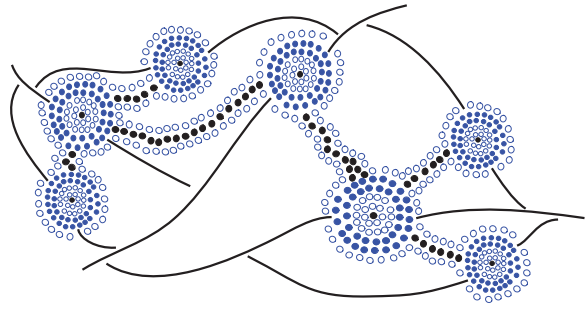
Chapter 3 Arrays, Linked Lists, and Recursion



Contents

3.1 Using Arrays

In this section, we explore a few applications of arrays—the concrete data structures introduced in Section 1.1.3 that access their entries using integer indices.

3.1.1 Storing Game Entries in an Array

The first application we study is for storing entries in an array; in particular, high score entries for a video game. Storing objects in arrays is a common use for arrays, and 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.

Let us begin by thinking about what we want to include in an object representing a high score entry. Obviously, one component to include is an integer representing the score itself, which we call *score*. Another useful thing to include is the name of the person earning this score, which we simply call *name*. We could go on from here, adding fields representing the date the score was earned or game statistics that led to that score. Let us keep our example simple, however, and just have two fields, *score* and *name*. The class structure is shown in Code Fragment 3.1.

```
class GameEntry {                                  // a game score entry
public:
  GameEntry(const string& n="", int s=0); // constructor
  string getName() const;                          // get player name
  int getScore() const;                            // get score
private:
  string name;                                     // player's name
  int score;                                       // player's score
};
```

Code Fragment 3.1: A C++ class representing a game entry.

In Code Fragment 3.2, we provide the definitions of the class constructor and two accessor member functions.

```
GameEntry::GameEntry(const string& n, int s) // constructor
  : name(n), score(s) { }
                                                   // accessors
string GameEntry::getName() const { return name; }
int GameEntry::getScore() const { return score; }
```

Code Fragment 3.2: GameEntry constructor and accessors.

A Class for High Scores

Let's now design a class, called Scores, to store our game-score information. We store the highest scores in an array *entries*. The maximum number of scores may vary from instance to instance, so we create a member variable, *maxEntries*, storing the desired maximum. Its value is specified when a Scores object is first constructed. In order to keep track of the actual number of entries, we define a member variable *numEntries*. It is initialized to zero, and it is updated as entries are added or removed. We provide a constructor, a destructor, a member function for adding a new score, and one for removing a score at a given index. The definition is given in Code Fragment 3.3.

```
class Scores {                              // stores game high scores
public:
  Scores(int maxEnt = 10);                  // constructor
  ~Scores();                                // destructor
  void add(const GameEntry& e);             // add a game entry
  GameEntry remove(int i)                   // remove the ith entry
      throw(IndexOutOfBounds);
private:
  int maxEntries;                           // maximum number of entries
  int numEntries;                           // actual number of entries
  GameEntry* entries;                       // array of game entries
};
```

Code Fragment 3.3: A C++ class for storing high game scores.

In Code Fragment 3.4, we present the class constructor, which allocates the desired amount of storage for the array using the "new" operator. Recall from Section 1.1.3 that C++ represents a dynamic array as a pointer to its first element, and this command returns such a pointer. The class destructor, ~Scores, deletes this array.

```
Scores::Scores(int maxEnt) {                // constructor
  maxEntries = maxEnt;                      // save the max size
  entries = new GameEntry[maxEntries];      // allocate array storage
  numEntries = 0;                           // initially no elements
}

Scores::~Scores() {                         // destructor
  delete[] entries;
}
```

Code Fragment 3.4: A C++ class GameEntry representing a game entry.

The entries that have been added to the array are stored in indices 0 through *numEntries* − 1. As more users play our video game, additional GameEntry objects

are copied into the array. This is done using the class's add member function, which we describe below. Only the highest *maxEntries* scores are retained. We also provide a member function, remove(i), which removes the entry at index i from the array. We assume that $0 \leq i \leq numEntries - 1$. If not, the remove function, throws an *IndexOutOfBounds* exception. We do not define this exception here, but it is derived from the class RuntimeException from Section 2.4.

In our design, we have chosen to order the GameEntry objects by their *score* values, from highest to lowest. (In Exercise C-3.2, we explore an alternative design in which entries are not ordered.) We illustrate an example of the data structure in Figure 3.1.

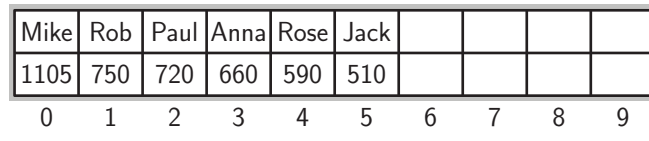


Figure 3.1: The *entries* array of length eight storing six GameEntry objects in the cells from index 0 to 5. Here *maxEntries* is 10 and *numEntries* is 6.

Insertion

Next, let us consider how to add a new GameEntry e to the array of high scores. In particular, let us consider how we might perform the following update operation on an instance of the Scores class.

add(e): Insert game entry e into the collection of high scores. If this causes the number of entries to exceed *maxEntries*, the smallest is removed.

The approach is to shift all the entries of the array whose scores are smaller than e's score to the right, in order to make space for the new entry. (See Figure 3.2.)

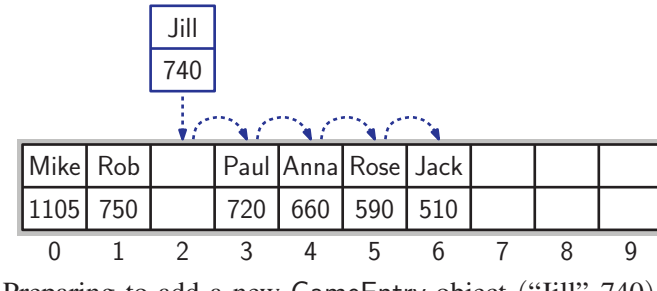


Figure 3.2: Preparing to add a new GameEntry object ("Jill", 740) to the *entries* array. In order to make room for the new entry, we shift all the entries with smaller scores to the right by one position.

Once we have identified the position in the *entries* array where the new game entry, e, belongs, we copy e into this position. (See Figure 3.3.)

Mike	Rob	Jill	Paul	Anna	Rose	Jack			
1105	750	740	720	660	590	510			
0	1	2	3	4	5	6	7	8	9

Figure 3.3: After adding the new entry at index 2.

The details of our algorithm for adding the new game entry e to the *entries* array are similar to this informal description and are given in Code Fragment 3.5. First, we consider whether the array is already full. If so, we check whether the score of the last entry in the array (which is at $entries[maxEntries - 1]$) is at least as large as e's score. If so, we can return immediately since e is not high enough to replace any of the existing highest scores. If the array is not yet full, we know that one new entry will be added, so we increment the value of *numEntries*. Next, we identify all the entries whose scores are smaller than e's and shift them one entry to the right. To avoid overwriting existing array entries, we start from the right end of the array and work to the left. The loop continues until we encounter an entry whose score is not smaller than e's, or we fall off the front end of the array. In either case, the new entry is added at index $i+1$.

```
void Scores::add(const GameEntry& e) {    // add a game entry
  int newScore = e.getScore();            // score to add
  if (numEntries == maxEntries) {         // the array is full
    if (newScore <= entries[maxEntries-1].getScore())
      return;                             // not high enough - ignore
  }
  else numEntries++;                      // if not full, one more entry

  int i = numEntries-2;                   // start with the next to last
  while ( i >= 0 && newScore > entries[i].getScore() ) {
    entries[i+1] = entries[i];            // shift right if smaller
    i--;
  }
  entries[i+1] = e;                       // put e in the empty spot
}
```

Code Fragment 3.5: C++ code for inserting a GameEntry object.

Check the code carefully to see that all the limiting cases have been handled correctly by the add function (for example, largest score, smallest score, empty array, full array). The number of times we perform the loop in this function depends on the number of entries that we need to shift. This is pretty fast if the number of entries is small. But if there are a lot to move, then this method could be fairly slow.

Object Removal

Suppose some hot shot plays our video game and gets his or her name on our high score list. In this case, we might want to have a function that lets us remove a game entry from the list of high scores. Therefore, let us consider how we might remove a GameEntry object from the *entries* array. That is, let us consider how we might implement the following operation:

remove(i): Remove and return the game entry e at index i in the *entries* array. If index i is outside the bounds of the *entries* array, then this function throws an exception; otherwise, the *entries* array is updated to remove the object at index i and all objects previously stored at indices higher than i are "shifted left" to fill in for the removed object.

Our implementation of remove is similar to that of add, but in reverse. To remove the entry at index i, we start at index i and move all the entries at indices higher than i one position to the left. (See Figure 3.4.)

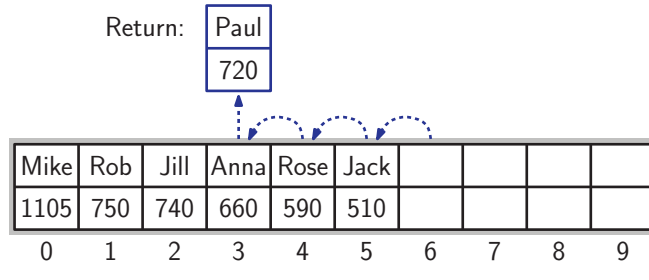


Figure 3.4: Removal of the entry ("Paul", 720) at index 3.

The code for performing the removal is presented in Code Fragment 3.6.

```
GameEntry Scores::remove(int i) throw(IndexOutOfBounds) {
  if ((i < 0) || (i >= numEntries))          // invalid index
    throw IndexOutOfBounds("Invalid index");
  GameEntry e = entries[i];                  // save the removed object
  for (int j = i+1; j < numEntries; j++)
    entries[j-1] = entries[j];               // shift entries left
  numEntries--;                              // one fewer entry
  return e;                                  // return the removed object
}
```

Code Fragment 3.6: C++ code for performing the remove operation.

The removal operation involves a few subtle points. In order to return the value of the removed game entry (let's call it e), we must first save e in a temporary variable. When we are done, the function will return this value. The shifting process starts at the position just following the removal, $j = i + 1$. We repeatedly copy the entry at index j to index $j - 1$, and then increment j, until coming to the last element of the set. Similar to the case of insertion, this left-to-right order is essential to avoid overwriting existing entries. To complete the function, we return a copy of the removed entry that was saved in e.

These functions 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 our simple array-based solution, and they may support many more operations. But studying the concrete array data structure, as we are doing now, is a great starting point for understanding these more sophisticated structures, since every data structure has to be implemented using concrete means.

3.1.2 Sorting an Array

In the previous subsection, we worked hard to show how we can add or remove objects at a certain index i in an array while keeping the previous order of the objects intact. In this section, we consider how to rearrange objects of an array that are ordered arbitrarily in ascending order. This is known as ***sorting***.

We study several sorting algorithms in this book, most of which appear in Chapter 11. As a warmup, we describe a simple sorting algorithm called ***insertion-sort***. In this case, we describe a specific version of the algorithm where the input is an array of comparable elements. We consider more general kinds of sorting algorithms later in this book.

We begin with a high-level outline of the insertion-sort algorithm. We start with the first element in the array. One element by itself is already sorted. Then we consider 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. We swap it leftward until it is in its proper order with the first two elements. We continue in this manner with each element of the array, swapping it leftward until it is in its proper position.

It is easy to see why this algorithm is called "insertion-sort"—each iteration of the algorithm inserts the next element into the current sorted part of the array, which was previously the subarray in front of that element. We may implement the above outline using two nested loops. The outer loop considers each element in the array in turn, and the inner loop moves that element to its proper location with the (sorted) subarray of elements that are to its left. We illustrate the resulting algorithm in Code Fragment 3.7.

This description is already quite close to actual C++ code. It indicates which

Algorithm InsertionSort(A):
Input: An array A of n comparable elements
Output: The array A with elements rearranged in nondecreasing order
for $i \leftarrow 1$ to $n-1$ **do**
{Insert $A[i]$ at its proper location in $A[0], A[1], \ldots, A[i-1]$}
$cur \leftarrow A[i]$
$j \leftarrow i-1$
while $j \geq 0$ and $A[j] > cur$ **do**
$A[j+1] \leftarrow A[j]$
$j \leftarrow j-1$
$A[j+1] \leftarrow cur$ {cur is now in the right place}

Code Fragment 3.7: Algorithmic description of the insertion-sort algorithm.

temporary variables are needed, how the loops are structured, and what decisions need to be made. We illustrate an example run in Figure 3.5.

We present C++ code for our insertion-sort algorithm in Code Fragment 3.8. We assume that the array to be sorted consists of elements of type **char**, but it is easy to generalize this to other data types. The array A in the algorithm is implemented as a **char** array. Recall that each array in C++ is represented as a pointer to its first element, so the parameter A is declared to be of type **char***. We also pass the size of the array in an integer parameter n. The rest is a straightforward translation of the description given in Code Fragment 3.7 into C++ syntax.

```
void insertionSort(char* A, int n) {          // sort an array of n characters
  for (int i = 1; i < n; i++) {               // insertion loop
    char cur = A[i];                          // current character to insert
    int j = i - 1;                            // start at previous character
    while ((j >= 0) && (A[j] > cur)) {        // while A[j] is out of order
      A[j + 1] = A[j];                        // move A[j] right
      j--;                                    // decrement j
    }
    A[j + 1] = cur;                           // this is the proper place for cur
  }
}
```

Code Fragment 3.8: C++ code implementing the insertion-sort algorithm.

An interesting thing happens in the insertion-sort algorithm if the array is already sorted. In this case, the inner loop 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. Indeed, the worst case arises if the initial array is given in descending order.

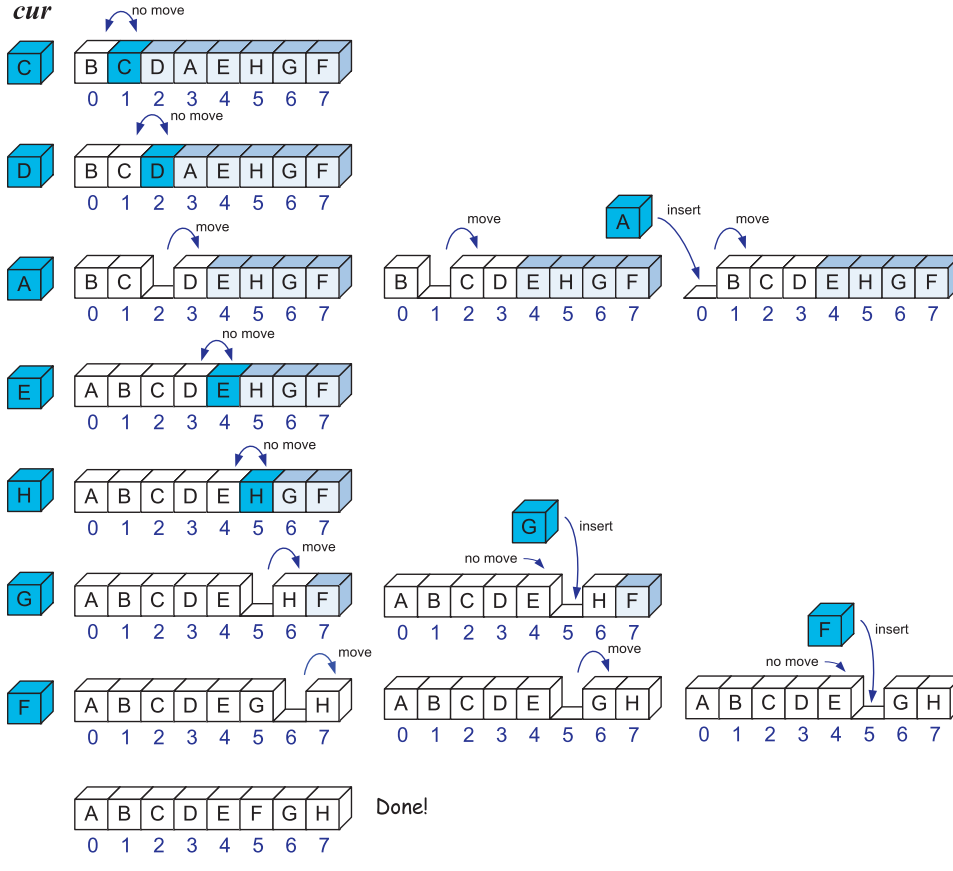


Figure 3.5: Execution of the insertion-sort algorithm on an array of eight characters. We show the completed (sorted) part of the array in white, and we color the next element that is being inserted into the sorted part of the array with light blue. We also highlight the character on the left, since it is stored in the *cur* variable. Each row corresponds to an iteration of the outer loop, and each copy of the array in a row corresponds to an iteration of the inner loop. Each comparison is shown with an arc. In addition, we indicate whether that comparison resulted in a move or not.

3.1.3 Two-Dimensional Arrays and Positional Games

Many computer games, be they strategy games, simulation games, or first-person conflict games, use a two-dimensional "board." Programs that deal with such ***positional games*** need 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 then maintain two-dimensional game boards, as well as perform other kinds of computations

involving data that is stored in rows and columns.

Arrays in C++ 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 C++—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 C++, we declare a two-dimensional array as follows:

```
int M[8][10];          // matrix with 8 rows and 10 columns
```

This statement creates a two-dimensional "array of arrays," M, which is 8×10, having 8 rows and 10 columns. That is, M is an array of length 8 such that each element of M is an array of length 10 of integers. (See Figure 3.6.)

	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 3.6: A two-dimensional integer array that has 8 rows and 10 columns. The value of $M[3][5]$ is 100 and the value of $M[6][2]$ is 632.

Given integer variables i and j, we could output the element of row i and column j (or equivalently, the jth element of the ith array) as follows:

```
cout << M[i][j];          // output element in row i column j
```

It is often a good idea to use symbolic constants to define the dimensions in order to make your intentions clearer to someone reading your program.

```
const int N_DAYS = 7;
const int N_HOURS = 24;
int schedule[N_DAYS][N_HOURS];
```

Dynamic Allocation of Matrices

If the dimensions of a two-dimensional array are not known in advance, it is necessary to allocate the array dynamically. This can be done by applying the method that we discussed earlier for allocating arrays in Section 1.1.3, but instead, we need to apply it to each individual row of the matrix.

For example, suppose that we wish to allocate an integer matrix with n rows and m columns. Each row of the matrix is an array of integers of length m. Recall that a dynamic array is represented as a pointer to its first element, so each row would be declared to be of type **int***. How do we group the individual rows together to form the matrix? The matrix is an array of row pointers. Since each row pointer is of type **int***, the matrix is of type **int****, that is, a pointer to a pointer of integers.

To generate our matrix, we first declare M to be of this type and allocate the n row pointers with the command "M = **new int***[n]." The ith row of the matrix is allocated with the statement "M[i] = **new int**[m]." In Code Fragment 3.9, we show how to do this given two integer variables n and m.

```
int** M = new int*[n];              // allocate an array of row pointers
for (int i = 0; i < n; i++)
  M[i] = new int[m];                // allocate the i-th row
```

Code Fragment 3.9: Allocating storage for a matrix as an array of arrays.

Once allocated, we can access its elements just as before, for example, as "$M[i][j]$." As shown in Code Fragment 3.10, deallocating the matrix involves reversing these steps. First, we deallocate each of the rows, one by one. We then deallocate the array of row pointers. Since we are deleting an array, we use the command "**delete**[]."

```
for (int i = 0; i < n; i++)
  delete[] M[i];                    // delete the i-th row
delete[] M;                         // delete the array of row pointers
```

Code Fragment 3.10: Deallocating storage for a matrix as an array of arrays.

Using STL Vectors to Implement Matrices

As we can see from the previous section, dynamic allocation of matrices is rather cumbersome. The STL vector class (recall Section 1.5.5) provides a much more elegant way to process matrices. We adapt the same approach as above by implementing a matrix as a vector of vectors. Each row of our matrix is declared as "vector<int>." Thus, the entire matrix is declared to be a vector of rows, that is, "vector<vector<int>>." Let us declare M to be of this type.

Letting n denote the desired number of rows in the matrix, the constructor call $M(n)$ allocates storage for the rows. However, this does not allocate the desired number of columns. The reason is that the default constructor is called for each row, and the default is to construct an empty array.

To fix this, we make use of a nice feature of the vector class constructor. There is an optional second argument, which indicates the value to use when initializing

each element of the vector. In our case, each element of M is a vector of m integers, that is, "vector<int>(m)." Thus, given integer variables n and m, the following code fragment generates an $n \times m$ matrix as a vector of vectors.

```
vector< vector<int> > M(n, vector<int>(m));
cout << M[i][j] << endl;
```

The space between vector<int> and the following ">" has been added to prevent ambiguity with the C++ input operator ">>." Because the STL vector class automatically takes care of deleting its members, we do not need to write a loop to explicitly delete the rows, as we needed with dynamic arrays.

Two-dimensional arrays have many applications. Next, we explore a simple application of two-dimensional arrays for implementing a positional game.

Tic-Tac-Toe

As most school children know, ***Tic-Tac-Toe*** is a game played on 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. (See Exercise P-7.11.)

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. For example, its 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 O. This encoding allows us to have a simple way of testing whether 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 give a complete C++ program for maintaining a Tic-Tac-Toe board for two players in Code Fragments 3.11 and 3.12. We show the resulting output in Figure 3.8. 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 project (see Exercise P-7.11).

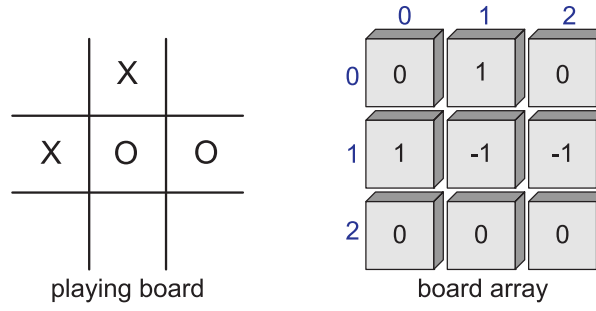


Figure 3.7: A Tic-Tac-Toe board and the array representing it.

```
#include <cstdlib>                          // system definitions
#include <iostream>                         // I/O definitions
using namespace std;                        // make std:: accessible

const int X = 1, O = -1, EMPTY = 0;         // possible marks
int board[3][3];                            // playing board
int currentPlayer;                          // current player (X or O)

void clearBoard() {                         // clear the board
  for (int i = 0; i < 3; i++)
    for (int j = 0; j < 3; j++)
      board[i][j] = EMPTY;                  // every cell is empty
  currentPlayer = X;                        // player X starts
}

void putMark(int i, int j) {                // mark row i column j
  board[i][j] = currentPlayer;              // mark with current player
  currentPlayer = -currentPlayer;           // switch players
}

bool isWin(int mark) {                      // is mark the winner?
  int win = 3*mark;                         // +3 for X and -3 for O
  return ((board[0][0] + board[0][1] + board[0][2] == win)     // row 0
       || (board[1][0] + board[1][1] + board[1][2] == win)     // row 1
       || (board[2][0] + board[2][1] + board[2][2] == win)     // row 2
       || (board[0][0] + board[1][0] + board[2][0] == win)     // column 0
       || (board[0][1] + board[1][1] + board[2][1] == win)     // column 1
       || (board[0][2] + board[1][2] + board[2][2] == win)     // column 2
       || (board[0][0] + board[1][1] + board[2][2] == win)     // diagonal
       || (board[2][0] + board[1][1] + board[0][2] == win));   // diagonal
}
```

Code Fragment 3.11: A C++ program for playing Tic-Tac-Toe between two players. (Continues in Code Fragment 3.12.)

```
int getWinner() {                                   // who wins? (EMPTY means tie)
  if (isWin(X)) return X;
  else if (isWin(O)) return O;
  else return EMPTY;
}

void printBoard() {                                 // print the board
  for (int i = 0; i < 3; i++) {
    for (int j = 0; j < 3; j++) {
      switch (board[i][j]) {
      case X:     cout << "X"; break;
      case O:     cout << "O"; break;
      case EMPTY: cout << " "; break;
      }
      if (j < 2) cout << "|";                       // column boundary
    }
    if (i < 2) cout << "\n-+-+-\n";                 // row boundary
  }
}

int main() {                                        // main program
  clearBoard();                                     // clear the board
  putMark(0,0);     putMark(1,1);                   // add the marks
  putMark(0,1);     putMark(0,2);
  putMark(2,0);     putMark(1,2);
  putMark(2,2);     putMark(2,1);
  putMark(1,0);
  printBoard();                                     // print the final board
  int winner = getWinner();
  if (winner != EMPTY)                              // print the winner
    cout << " " << (winner == X ? 'X' : 'O') << " wins" << endl;
  else
    cout << " Tie" << endl;
  return EXIT_SUCCESS;
}
```

Code Fragment 3.12: A C++ program for playing Tic-Tac-Toe between two players. (Continued from Code Fragment 3.11.)

```
X|X|O
-+-+-
X|O|O
-+-+-
X|O|X  X wins
```

Figure 3.8: Output of the Tic-Tac-Toe program.

3.2 Singly Linked Lists

In the previous section, we presented the array data structure and discussed some of its applications. Arrays are nice and simple for storing things in a certain order, but they have drawbacks. They are not very adaptable. For instance, we have to fix the size n of an array in advance, which makes resizing an array difficult. (This drawback is remedied in STL vectors.) Insertions and deletions are difficult because elements need to be shifted around to make space for insertion or to fill empty positions after deletion. In this section, we explore an important alternate implementation of sequence, known as the singly linked list.

A ***linked list***, in its simplest form, is a collection of ***nodes*** that together form a linear ordering. As in the children's game "Follow the Leader," each node stores a pointer, called *next*, to the next node of the list. In addition, each node stores its associated element. (See Figure 3.9.)

Figure 3.9: Example of a singly linked list of airport codes. The *next* pointers are shown as arrows. The null pointer is denoted by $\emptyset$.

The *next* pointer inside a node is a ***link*** or ***pointer*** to the next node of the list. Moving from one node to another by following a *next* reference is known as ***link hopping*** or ***pointer hopping***. The first and last nodes of a linked list are called the ***head*** and ***tail*** of the list, respectively. Thus, we can link-hop through the list, starting at the head and ending at the tail. We can identify the tail as the node having a null *next* reference. The structure is called a ***singly linked list*** because each node stores a single link.

Like an array, a singly linked list maintains its elements in a certain order, as determined by the chain of *next* links. Unlike an array, a singly linked list does not have a predetermined fixed size. It can be resized by adding or removing nodes.

3.2.1 Implementing a Singly Linked List

Let us implement a singly linked list of strings. We first define a class StringNode shown in Code Fragment 3.13. The node stores two values, the member *elem* stores the element stored in this node, which in this case is a character string. (Later, in Section 3.2.4, we describe how to define nodes that can store arbitrary types of elements.) The member *next* stores a pointer to the next node of the list. We make the linked list class a friend, so that it can access the node's private members.

```
class StringNode {                          // a node in a list of strings
private:
  string elem;                              // element value
  StringNode* next;                         // next item in the list

  friend class StringLinkedList;            // provide StringLinkedList access
};
```

Code Fragment 3.13: A node in a singly linked list of strings.

In Code Fragment 3.14, we define a class StringLinkedList for the actual linked list. It supports a number of member functions, including a constructor and destructor and functions for insertion and deletion. Their implementations are presented later. Its private data consists of a pointer to the head node of the list.

```
class StringLinkedList {                    // a linked list of strings
public:
  StringLinkedList();                       // empty list constructor
  ~StringLinkedList();                      // destructor
  bool empty() const;                       // is list empty?
  const string& front() const;              // get front element
  void addFront(const string& e);           // add to front of list
  void removeFront();                       // remove front item list
private:
  StringNode* head;                         // pointer to the head of list
};
```

Code Fragment 3.14: A class definition for a singly linked list of strings.

A number of simple member functions are shown in Code Fragment 3.15. The list constructor creates an empty list by setting the head pointer to NULL. The destructor repeatedly removes elements from the list. It exploits the fact that the function remove (presented below) destroys the node that it removes. To test whether the list is empty, we simply test whether the head pointer is NULL.

```
StringLinkedList::StringLinkedList()                    // constructor
  : head(NULL) { }

StringLinkedList::~StringLinkedList()                   // destructor
  { while (!empty()) removeFront(); }

bool StringLinkedList::empty() const                    // is list empty?
  { return head == NULL; }

const string& StringLinkedList::front() const           // get front element
  { return head->elem; }
```

Code Fragment 3.15: Some simple member functions of class StringLinkedList.

3.2.2 Insertion to the Front of a Singly Linked List

We can easily insert an element at the head of a singly linked list. We first create a new node, and set its *elem* value to the desired string and set its *next* link to point to the current head of the list. We then set *head* to point to the new node. The process is illustrated in Figure 3.10.

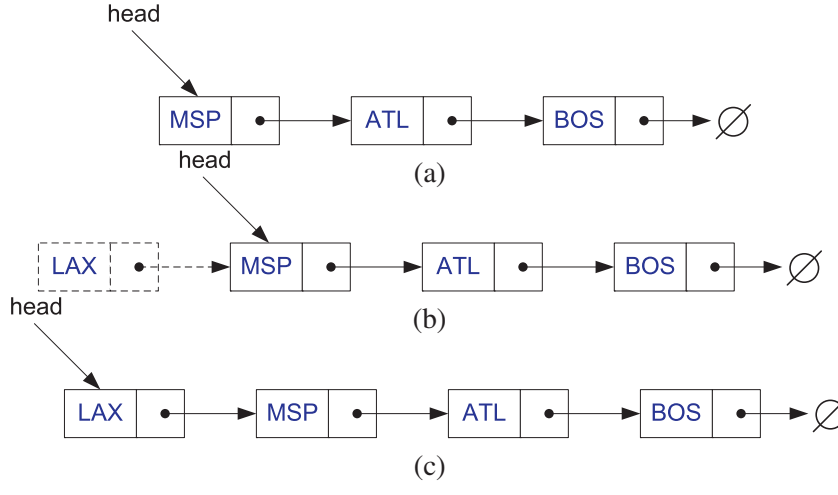


Figure 3.10: Insertion of an element at the head of a singly linked list: (a) before the insertion; (b) creation of a new node; (c) after the insertion.

An implementation is shown in Code Fragment 3.16. Note that access to the private members *elem* and *next* of the StringNode class would normally be prohibited, but it is allowed here because StringLinkedList was declared to be a friend of StringNode.

```
void StringLinkedList::addFront(const string& e) {  // add to front of list
  StringNode* v = new StringNode;                   // create new node
  v->elem = e;                                      // store data
  v->next = head;                                   // head now follows v
  head = v;                                         // v is now the head
}
```

Code Fragment 3.16: Insertion to the front of a singly linked list.

3.2.3 Removal from the Front of a Singly Linked List

Next, we consider how to remove an element from the front of a singly linked list. We essentially undo the operations performed for insertion. We first save a pointer

to the old head node and advance the head pointer to the next node in the list. We then delete the old head node. This operation is illustrated in Figure 3.11.

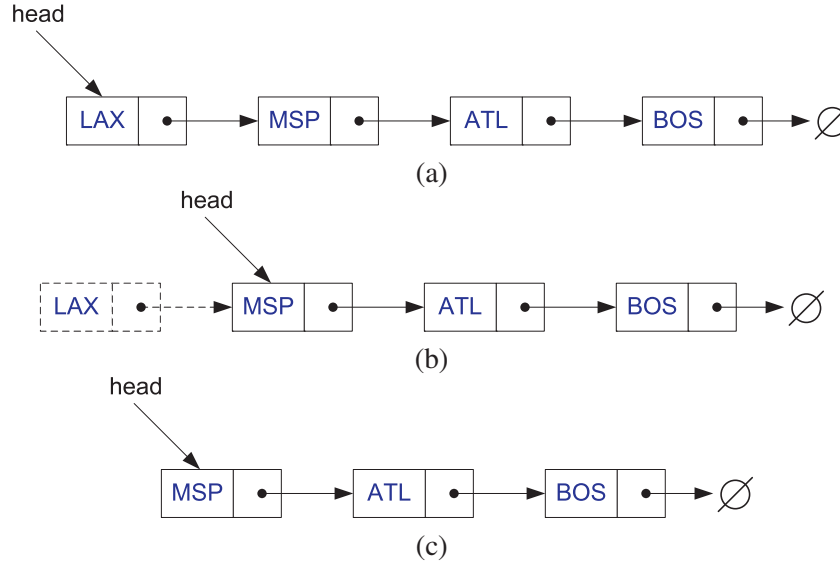


Figure 3.11: Removal of an element at the head of a singly linked list: (a) before the removal; (b) "linking out" the old new node; (c) after the removal.

An implementation of this operation is provided in Code Fragment 3.17. We assume that the user has checked that the list is nonempty before applying this operation. (A more careful implementation would throw an exception if the list were empty.) The function deletes the node in order to avoid any memory leaks. We do not return the value of the deleted node. If its value is desired, we can call the front function prior to the removal.

```
void StringLinkedList::removeFront() {          // remove front item
  StringNode* old = head;                       // save current head
  head = old->next;                             // skip over old head
  delete old;                                   // delete the old head
}
```

Code Fragment 3.17: Removal from the front of a singly linked list.

It is noteworthy that we cannot as easily delete the last node of a singly linked list, even if we had a pointer to it. In order to delete a node, we need to update the *next* link of the node immediately ***preceding*** the deleted node. Locating this node involves traversing the entire list and could take a long time. (We remedy this in Section 3.3 when we discuss doubly linked lists.)

3.2.4 Implementing a Generic Singly Linked List

The implementation of the singly linked list given in Section 3.2.1 assumes that the element type is a character string. It is easy to convert the implementation so that it works for an arbitrary element type through the use of C++'s template mechanism. The resulting generic singly linked list class is called SLinkedList.

We begin by presenting the node class, called SNode, in Code Fragment 3.18. The element type associated with each node is parameterized by the type variable E. In contrast to our earlier version in Code Fragment 3.13, references to the data type "string" have been replaced by "E." When referring to our templated node and list class, we need to include the suffix "<E>." For example, the class is SLinkedList<E> and the associated node is SNode<E>.

```
template <typename E>
class SNode {                              // singly linked list node
private:
  E elem;                                  // linked list element value
  SNode<E>* next;                          // next item in the list
  friend class SLinkedList<E>;             // provide SLinkedList access
};
```

Code Fragment 3.18: A node in a generic singly linked list.

The generic list class is presented in Code Fragment 3.19. As above, references to the specific element type "string" have been replaced by references to the generic type parameter "E." To keep things simple, we have omitted housekeeping functions such as a copy constructor.

```
template <typename E>
class SLinkedList {                        // a singly linked list
public:
  SLinkedList();                           // empty list constructor
  ~SLinkedList();                          // destructor
  bool empty() const;                      // is list empty?
  const E& front() const;                  // return front element
  void addFront(const E& e);               // add to front of list
  void removeFront();                      // remove front item list
private:
  SNode<E>* head;                          // head of the list
};
```

Code Fragment 3.19: A class definition for a generic singly linked list.

In Code Fragment 3.20, we present the class member functions. Note the similarity with Code Fragments 3.15 through 3.17. Observe that each definition is prefaced by the template specifier **template <typename** E>.

```
template <typename E>
SLinkedList<E>::SLinkedList()                    // constructor
  : head(NULL) { }

template <typename E>
bool SLinkedList<E>::empty() const               // is list empty?
  { return head == NULL; }

template <typename E>
const E& SLinkedList<E>::front() const           // return front element
  { return head->elem; }

template <typename E>
SLinkedList<E>::~SLinkedList()                   // destructor
  { while (!empty()) removeFront(); }

template <typename E>
void SLinkedList<E>::addFront(const E& e) {      // add to front of list
  SNode<E>* v = new SNode<E>;                    // create new node
  v->elem = e;                                   // store data
  v->next = head;                                // head now follows v
  head = v;                                      // v is now the head
}

template <typename E>
void SLinkedList<E>::removeFront() {             // remove front item
  SNode<E>* old = head;                          // save current head
  head = old->next;                              // skip over old head
  delete old;                                    // delete the old head
}
```

Code Fragment 3.20: Other member functions for a generic singly linked list.

We can generate singly linked lists of various types by simply setting the template parameter as desired as shown in the following code fragment.

```
SLinkedList<string> a;                           // list of strings
a.addFront("MSP");
// ...
SLinkedList<int> b;                              // list of integers
b.addFront(13);
```

Code Fragment 3.21: Examples using the generic singly linked list class.

Because templated classes carry a relatively high notational burden, we often sacrifice generality for simplicity, and avoid the use of templated classes in some of our examples.

3.3 Doubly Linked Lists

As we saw in the previous section, removing an element at the tail of a singly linked list is not easy. Indeed, it is time consuming to remove any node other than the head in a singly linked list, since we do not have a quick way of accessing the node immediately preceding the one we want to remove. There are many applications where we do not have quick access to such a predecessor node. For such applications, it would be nice to have a way of going both directions in a linked list.

There is a type of linked list that allows us to go in both directions—forward and reverse—in a linked list. It is the ***doubly linked*** list. In addition to its element member, a node in a doubly linked list stores two pointers, a *next* link and a *prev* link, which point to the next node in the list and the previous node in the list, respectively. Such lists allow for a great variety of quick update operations, including efficient insertion and removal at any given position.

Header and Trailer Sentinels

To simplify programming, it is convenient to add special nodes at both ends of a doubly linked list: a ***header*** node just before the head of the list, and a ***trailer*** node just after the tail of the list. These "dummy" or ***sentinel*** nodes do not store any elements. They provide quick access to the first and last nodes of the list. In particular, the header's *next* pointer points to the first node of the list, and the *prev* pointer of the trailer node points to the last node of the list. An example is shown in Figure 3.12.

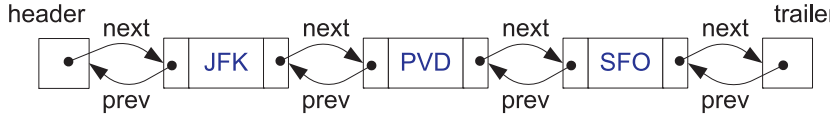


Figure 3.12: A doubly linked list with sentinels, *header* and *trailer*, marking the ends of the list. An empty list would have these sentinels pointing to each other. We do not show the null *prev* pointer for the *header* nor do we show the null *next* pointer for the *trailer*.

3.3.1 Insertion into a Doubly Linked List

Because of its double link structure, it is possible to insert a node at any position within a doubly linked list. Given a node v of a doubly linked list (which could possibly be the header, but not the trailer), let z be a new node that we wish to insert

immediately after v. Let w the be node following v, that is, w is the node pointed to by v's next link. (This node exists, since we have sentinels.) To insert z after v, we link it into the current list, by performing the following operations:

- Make z's *prev* link point to v
- Make z's *next* link point to w
- Make w's *prev* link point to z
- Make v's *next* link point to z

This process is illustrated in Figure 3.13, where v points to the node JFK, w points to PVD, and z points to the new node BWI. Observe that this process works if v is any node ranging from the header to the node just prior to the trailer.

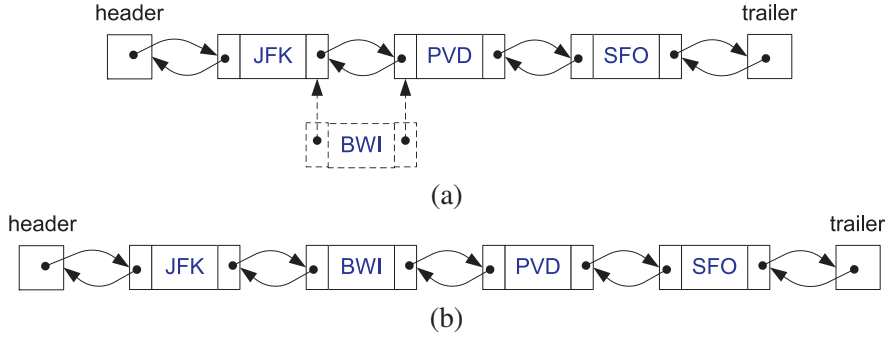


Figure 3.13: Adding a new node after the node storing JFK: (a) creating a new node with element BWI and linking it in; (b) after the insertion.

3.3.2 Removal from a Doubly Linked List

Likewise, it is easy to remove a node v from a doubly linked list. Let u be the node just prior to v, and w be the node just following v. (These nodes exist, since we have sentinels.) To remove node v, we simply have u and w point to each other instead of to v. We refer to this operation as the ***linking out*** of v. We perform the following operations.

- Make w's *prev* link point to u
- Make u's *next* link point to w
- Delete node v

This process is illustrated in Figure 3.14, where v is the node PVD, u is the node JFK, and w is the node SFO. Observe that this process works if v is any node from the header to the tail node (the node just prior to the trailer).

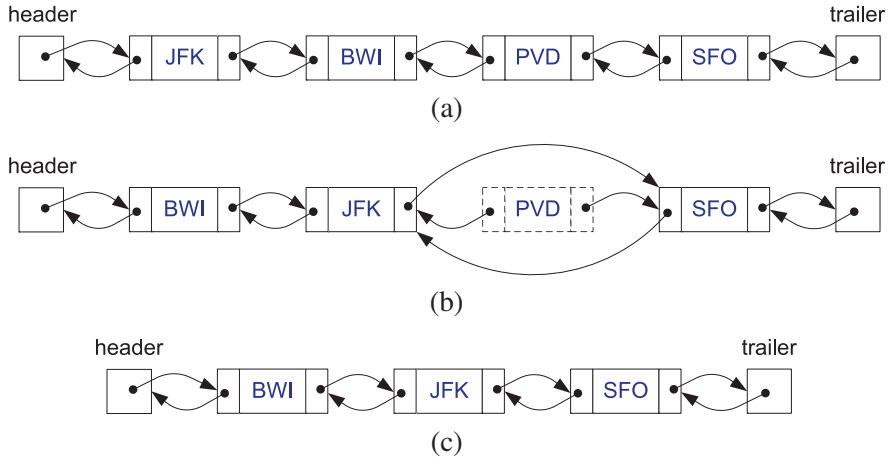


Figure 3.14: Removing the node storing PVD: (a) before the removal; (b) linking out the old node; (c) after node deletion.

3.3.3 A C++ Implementation

Let us consider how to implement a doubly linked list in C++. First, we present a C++ class for a node of the list in Code Fragment 3.22. To keep the code simple, we have chosen not to derive a templated class as we did in Section 3.2.1 for singly linked lists. Instead, we provide a **typedef** statement that defines the element type, called Elem. We define it to be a string, but any other type could be used instead. Each node stores an element. It also contains pointers to both the previous and next nodes of the list. We declare DLinkedList to be a friend, so it can access the node's private members.

```
typedef string Elem;                    // list element type
class DNode {                           // doubly linked list node
private:
  Elem elem;                            // node element value
  DNode* prev;                          // previous node in list
  DNode* next;                          // next node in list
  friend class DLinkedList;             // allow DLinkedList access
};
```

Code Fragment 3.22: C++ implementation of a doubly linked list node.

Next, we present the definition of the doubly linked list class, DLinkedList, in Code Fragment 3.23. In addition to a constructor and destructor, the public members consist of a function that indicates whether the list is currently empty

(meaning that it has no nodes other than the sentinels) and accessors to retrieve the front and back elements. We also provide methods for inserting and removing elements from the front and back of the list. There are two private data members, *header* and *trailer*, which point to the sentinels. Finally, we provide two protected utility member functions, add and remove. They are used internally by the class and by its subclasses, but they cannot be invoked from outside the class.

```
class DLinkedList {                          // doubly linked list
public:
  DLinkedList();                             // constructor
  ~DLinkedList();                            // destructor
  bool empty() const;                        // is list empty?
  const Elem& front() const;                 // get front element
  const Elem& back() const;                  // get back element
  void addFront(const Elem& e);              // add to front of list
  void addBack(const Elem& e);               // add to back of list
  void removeFront();                        // remove from front
  void removeBack();                         // remove from back
private:                                     // local type definitions
  DNode* header;                             // list sentinels
  DNode* trailer;
protected:                                   // local utilities
  void add(DNode* v, const Elem& e);         // insert new node before v
  void remove(DNode* v);                     // remove node v
};
```

Code Fragment 3.23: Implementation of a doubly linked list class.

Let us begin by presenting the class constructor and destructor as shown in Code Fragment 3.24. The constructor creates the sentinel nodes and sets each to point to the other, and the destructor removes all but the sentinel nodes.

```
DLinkedList::DLinkedList() {                 // constructor
  header = new DNode;                        // create sentinels
  trailer = new DNode;
  header->next = trailer;                    // have them point to each other
  trailer->prev = header;
}

DLinkedList::~DLinkedList() {                // destructor
  while (!empty()) removeFront();            // remove all but sentinels
  delete header;                             // remove the sentinels
  delete trailer;
}
```

Code Fragment 3.24: Class constructor and destructor.

Next, in Code Fragment 3.25 we show the basic class accessors. To determine whether the list is empty, we check that there is no node between the two sentinels. We do this by testing whether the trailer follows immediately after the header. To access the front element of the list, we return the element associated with the node that follows the list header. To access the back element, we return the element associated with node that precedes the trailer. Both operations assume that the list is nonempty. We could have enhanced these functions by throwing an exception if an attempt is made to access the front or back of an empty list, just as we did in Code Fragment 3.6.

```
bool DLinkedList::empty() const                // is list empty?
  { return (header->next == trailer); }

const Elem& DLinkedList::front() const         // get front element
  { return header->next->elem; }

const Elem& DLinkedList::back() const          // get back element
  { return trailer->prev->elem; }
```

Code Fragment 3.25: Accessor functions for the doubly linked list class.

In Section 3.3.1, we discussed how to insert a node into a doubly linked list. The local utility function add, which is shown in Code Fragment 3.26, implements this operation. In order to add a node to the front of the list, we create a new node, and insert it immediately after the header, or equivalently, immediately before the node that follows the header. In order to add a new node to the back of the list, we create a new node, and insert it immediately before the trailer.

```
                                               // insert new node before v
void DLinkedList::add(DNode* v, const Elem& e) {
  DNode* u = new DNode; u->elem = e;           // create a new node for e
  u->next = v;                                 // link u in between v
  u->prev = v->prev;                           // ...and v->prev
  v->prev->next = v->prev = u;
}

void DLinkedList::addFront(const Elem& e)      // add to front of list
  { add(header->next, e); }

void DLinkedList::addBack(const Elem& e)       // add to back of list
  { add(trailer, e); }
```

Code Fragment 3.26: Inserting a new node into a doubly linked list. The protected utility function add inserts a node z before an arbitrary node v. The public member functions addFront and addBack both invoke this utility function.

Observe that the above code works even if the list is empty (meaning that the only nodes are the header and trailer). For example, if addBack is invoked on an empty list, then the value of *trailer->prev* is a pointer to the list header. Thus, the node is added between the header and trailer as desired. One of the major advantages of providing sentinel nodes is to avoid handling of special cases, which would otherwise be needed.

Finally, let us discuss deletion. In Section 3.3.2, we showed how to remove an arbitrary node from a doubly linked list. In Code Fragment 3.27, we present local utility function remove, which performs the operation. In addition to linking out the node, it also deletes the node. The public member functions removeFront and removeBack are implemented by deleting the nodes immediately following the header and immediately preceding the trailer, respectively.

```
void DLinkedList::remove(DNode* v) {      // remove node v
  DNode* u = v->prev;                     // predecessor
  DNode* w = v->next;                     // successor
  u->next = w;                            // unlink v from list
  w->prev = u;
  delete v;
}

void DLinkedList::removeFront()           // remove from font
  { remove(header->next); }

void DLinkedList::removeBack()            // remove from back
  { remove(trailer->prev); }
```

Code Fragment 3.27: Removing a node from a doubly linked list. The local utility function remove removes the node *v*. The public member functions removeFront and removeBack invoke this utility function.

There are many more features that we could have added to our simple implementation of a doubly linked list. Although we have provided access to the ends of the list, we have not provided any mechanism for accessing or modifying elements in the middle of the list. Later, in Chapter 6, we discuss the concept of iterators, which provides a mechanism for accessing arbitrary elements of a list.

We have also performed no error checking in our implementation. It is the user's responsibility not to attempt to access or remove elements from an empty list. In a more robust implementation of a doubly linked list, we would design the member functions front, back, removeFront, and removeBack to throw an exception when an attempt is made to perform one of these functions on an empty list. Nonetheless, this simple implementation illustrates how easy it is to manipulate this useful data structure.

3.4 Circularly Linked Lists and List Reversal

In this section, we study some applications and extensions of linked lists.

3.4.1 Circularly Linked Lists

A ***circularly linked list*** has the same kind of nodes as a singly linked list. That is, each node in a circularly linked list has a next pointer and an element value. But, rather than having a head or tail, the nodes of a circularly linked list are linked into a cycle. If we traverse the nodes of a circularly linked list from any node by following **next** pointers, we eventually visit all the nodes and cycle back to the node from which we started.

Even though a circularly linked list has no beginning or end, we nevertheless need some node to be marked as a special node, which we call the ***cursor***. The cursor node allows us to have a place to start from if we ever need to traverse a circularly linked list.

There are two positions of particular interest in a circular list. The first is the element that is referenced by the cursor, which is called the ***back***, and the element immediately following this in the circular order, which is called the ***front***. Although it may seem odd to think of a circular list as having a front and a back, observe that, if we were to cut the link between the node referenced by the cursor and this node's immediate successor, the result would be a singly linked list from the front node to the back node.

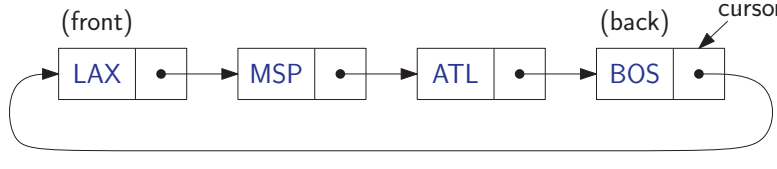


Figure 3.15: A circularly linked list. The node referenced by the cursor is called the back, and the node immediately following is called the front.

We define the following functions for a circularly linked list:

front(): Return the element referenced by the cursor; an error results if the list is empty.

back(): Return the element immediately after the cursor; an error results if the list is empty.

advance(): Advance the cursor to the next node in the list.

add(e): Insert a new node with element e immediately after the cursor; if the list is empty, then this node becomes the cursor and its *next* pointer points to itself.

remove(): Remove the node immediately after the cursor (not the cursor itself, unless it is the only node); if the list becomes empty, the cursor is set to *null*.

In Code Fragment 3.28, we show a C++ implementation of a node of a circularly linked list, assuming that each node contains a single string. The node structure is essentially identical to that of a singly linked list (recall Code Fragment 3.13). To keep the code simple, we have not implemented a templated class. Instead, we provide a **typedef** statement that defines the element type Elem to be the base type of the list, which in this case is a string.

```
typedef string Elem;                    // element type
class CNode {                           // circularly linked list node
private:
  Elem elem;                            // linked list element value
  CNode* next;                          // next item in the list

  friend class CircleList;              // provide CircleList access
};
```

Code Fragment 3.28: A node of a circularly linked list.

Next, in Code Fragment 3.29, we present the class definition for a circularly linked list called CircleList. In addition to the above functions, the class provides a constructor, a destructor, and a function to detect whether the list is empty. The private member consists of the cursor, which points to some node of the list.

```
class CircleList {                      // a circularly linked list
public:
  CircleList();                         // constructor
  ~CircleList();                        // destructor
  bool empty() const;                   // is list empty?
  const Elem& front() const;            // element at cursor
  const Elem& back() const;             // element following cursor
  void advance();                       // advance cursor
  void add(const Elem& e);              // add after cursor
  void remove();                        // remove node after cursor
private:
  CNode* cursor;                        // the cursor
};
```

Code Fragment 3.29: Implementation of a circularly linked list class.

Code Fragment 3.30 presents the class's constructor and destructor. The constructor generates an empty list by setting the cursor to NULL. The destructor iteratively removes nodes until the list is empty. We exploit the fact that the member function remove (given below) deletes the node that it removes.

```
CircleList::CircleList()                    // constructor
  : cursor(NULL) { }
CircleList::~CircleList()                   // destructor
  { while (!empty()) remove(); }
```

Code Fragment 3.30: The constructor and destructor.

We present a number of simple member functions in Code Fragment 3.31. To determine whether the list is empty, we test whether the cursor is NULL. The advance function advances the cursor to the next element.

```
bool CircleList::empty() const              // is list empty?
  { return cursor == NULL; }
const Elem& CircleList::back() const        // element at cursor
  { return cursor->elem; }
const Elem& CircleList::front() const       // element following cursor
  { return cursor->next->elem; }
void CircleList::advance()                  // advance cursor
  { cursor = cursor->next; }
```

Code Fragment 3.31: Simple member functions.

Next, let us consider insertion. Recall that insertions to the circularly linked list occur ***after*** the cursor. We begin by creating a new node and initializing its data member. If the list is empty, we create a new node that points to itself. We then direct the cursor to point to this element. Otherwise, we link the new node just after the cursor. The code is presented in Code Fragment 3.32.

```
void CircleList::add(const Elem& e) {       // add after cursor
  CNode* v = new CNode;                     // create a new node
  v->elem = e;
  if (cursor == NULL) {                     // list is empty?
    v->next = v;                            // v points to itself
    cursor = v;                             // cursor points to v
  }
  else {                                    // list is nonempty?
    v->next = cursor->next;                 // link in v after cursor
    cursor->next = v;
  }
}
```

Code Fragment 3.32: Inserting a node just after the cursor of a circularly linked list.

Finally, we consider removal. We assume that the user has checked that the list is nonempty before invoking this function. (A more careful implementation would throw an exception if the list is empty.) There are two cases. If this is the last node of the list (which can be tested by checking that the node to be removed points to itself) we set the cursor to NULL. Otherwise, we link the cursor's next pointer to skip over the removed node. We then delete the node. The code is presented in Code Fragment 3.33.

```
void CircleList::remove() {                    // remove node after cursor
  CNode* old = cursor->next;                   // the node being removed
  if (old == cursor)                           // removing the only node?
    cursor = NULL;                             // list is now empty
  else
    cursor->next = old->next;                  // link out the old node
  delete old;                                  // delete the old node
}
```

Code Fragment 3.33: Removing the node following the cursor.

To keep the code simple, we have omitted error checking. In front, back, and advance, we should first test whether the list is empty, since otherwise the cursor pointer will be NULL. In the first two cases, we should throw some sort of exception. In the case of advance, if the list is empty, we can simply return.

Maintaining a Playlist for a Digital Audio Player

To help illustrate the use of our CircleList implementation of the circularly linked list, let us consider how to build a simple interface for maintaining a playlist for a digital audio player, also known as an MP3 player. The songs of the player are stored in a circular list. The cursor points to the current song. By advancing the cursor, we can move from one song to the next. We can also add new songs and remove songs by invoking the member functions insert and remove, respectively. Of course, a complete implementation would need to provide a method for playing the current song, but our purpose is to illustrate how the circularly linked list can be applied to this task.

To make this more concrete, suppose that you have a friend who loves retro music, and you want to create a playlist of songs from the bygone Disco Era. The main program is presented Code Fragment 3.34. We declare an object *playList* to be a CircleList. The constructor creates an empty playlist. We proceed to add three songs, "Stayin Alive," "Le Freak," and "Jive Talkin." The comments on the right show the current contents of the list in square brackets. The first entry of the list is the element immediately following the cursor (which is where insertion and removal occur), and the last entry in the list is cursor (which is indicated with an asterisk).

Suppose that we decide to replace "Stayin Alive" with "Disco Inferno." We advance the cursor twice so that "Stayin Alive" comes immediately after the cursor. We then remove this entry and insert its replacement.

```
int main() {
  CircleList playList;                  // []
  playList.add("Stayin Alive");         // [Stayin Alive*]
  playList.add("Le Freak");             // [Le Freak, Stayin Alive*]
  playList.add("Jive Talkin");          // [Jive Talkin, Le Freak, Stayin Alive*]

  playList.advance();                   // [Le Freak, Stayin Alive, Jive Talkin*]
  playList.advance();                   // [Stayin Alive, Jive Talkin, Le Freak*]
  playList.remove();                    // [Jive Talkin, Le Freak*]
  playList.add("Disco Inferno");        // [Disco Inferno, Jive Talkin, Le Freak*]
  return EXIT_SUCCESS;
}
```

Code Fragment 3.34: Using the CircleList class to implement a playlist for a digital audio player.

3.4.2 Reversing a Linked List

As another example of the manipulation of linked lists, we present a simple function for reversing the elements of a doubly linked list. Given a list L, our approach involves first copying the contents of L in reverse order into a temporary list T, and then copying the contents of T back into L (but without reversing).

To achieve the initial reversed copy, we repeatedly extract the first element of L and copy it to the front of T. (To see why this works, observe that the later an element appears in L, the earlier it will appear in T.) To copy the contents of T back to L, we repeatedly extract elements from the front of T, but this time we copy each one to the back of list L. Our C++ implementation is presented in Code Fragment 3.35.

```
void listReverse(DLinkedList& L) {          // reverse a list
  DLinkedList T;                            // temporary list
  while (!L.empty()) {                      // reverse L into T
    string s = L.front();  L.removeFront();
    T.addFront(s);
  }
  while (!T.empty()) {                      // copy T back to L
    string s = T.front();  T.removeFront();
    L.addBack(s);
  }
}
```

Code Fragment 3.35: A function that reverses the contents of a doubly linked list L.

3.5 Recursion

We have seen that repetition can be achieved by writing loops, such as **for** loops and **while** loops. Another way to achieve repetition is through ***recursion***, which occurs when a function refers to itself in its own definition. We have seen examples of functions calling other functions, so it should come as no surprise that most modern programming languages, including C++, allow a function to call itself. In this section, we see why this capability provides an elegant and powerful alternative for performing repetitive tasks.

The Factorial Function

To illustrate recursion, let us begin with a simple example of computing the value of the ***factorial function***. The factorial of a positive integer n, denoted $n!$, is defined as the product of the integers from 1 to n. If $n = 0$, then $n!$ is defined as 1 by convention. More formally, for any integer $n \geq 0$,

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ n \cdot (n-1) \cdot (n-2) \cdots 3 \cdot 2 \cdot 1 & \text{if } n \geq 1. \end{cases}$$

For example, $5! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1 = 120$. To make the connection with functions clearer, we use the notation $\mathsf{factorial}(n)$ to denote $n!$.

The factorial function can be defined in a manner that suggests a recursive formulation. To see this, observe that

$$\mathsf{factorial}(5) = 5 \cdot (4 \cdot 3 \cdot 2 \cdot 1) = 5 \cdot \mathsf{factorial}(4).$$

Thus, we can define $\mathsf{factorial}(5)$ in terms of $\mathsf{factorial}(4)$. In general, for a positive integer n, we can define $\mathsf{factorial}(n)$ to be $n \cdot \mathsf{factorial}(n-1)$. This leads to the following ***recursive definition***

$$\mathsf{factorial}(n) = \begin{cases} 1 & \text{if } n = 0 \\ n \cdot \mathsf{factorial}(n-1) & \text{if } n \geq 1. \end{cases}$$

This definition is typical of many recursive definitions. First, it contains one or more ***base cases***, which are defined nonrecursively in terms of fixed quantities. In this case, $n = 0$ is the base case. It also contains one or more ***recursive cases***, which are defined by appealing to the definition of the function being defined. Observe that there is no circularity in this definition because each time the function is invoked, its argument is smaller by one.

A Recursive Implementation of the Factorial Function

Let us consider a C++ implementation of the factorial function shown in Code Fragment 3.36 under the name recursiveFactorial. Notice that no looping was needed here. The repeated recursive invocations of the function take the place of looping.

```
int recursiveFactorial(int n) {                  // recursive factorial function
  if (n == 0) return 1;                          // basis case
  else return n * recursiveFactorial(n−1);       // recursive case
}
```

Code Fragment 3.36: A recursive implementation of the factorial function.

We can illustrate the execution of a recursive function definition by means of a ***recursion trace***. Each entry of the trace corresponds to a recursive call. Each new recursive function call is indicated by an arrow to the newly called function. When the function returns, an arrow showing this return is drawn, and the return value may be indicated with this arrow. An example of a trace is shown in Figure 3.16.

What is the advantage of using recursion? Although the recursive implementation of the factorial function is somewhat simpler than the iterative version, in this case there is no compelling reason for preferring recursion over iteration. For some problems, however, a recursive implementation can be significantly simpler and easier to understand than an iterative implementation. Such an example follows.

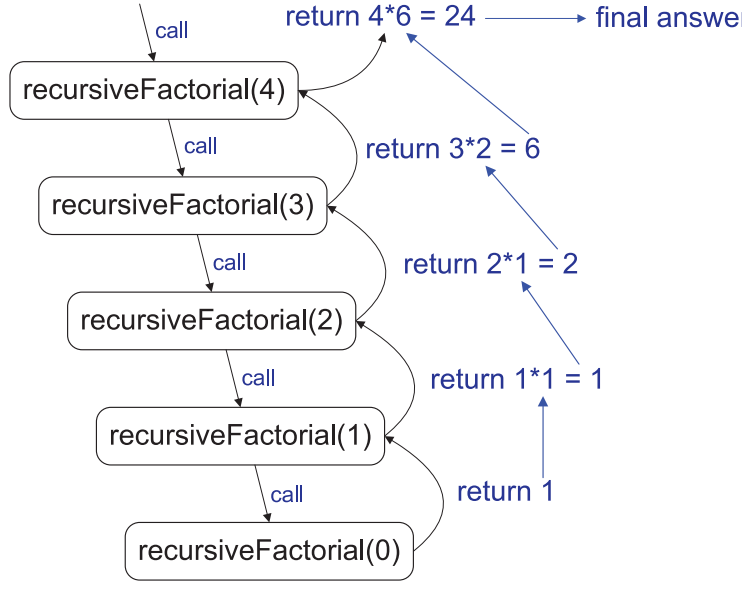


Figure 3.16: A recursion trace for the call recursiveFactorial(4).

Drawing an English Ruler

As a more complex example of the use of recursion, consider how to draw the markings of a typical English ruler. Such a ruler is broken into intervals, and each interval consists of a set of ***ticks***, placed at intervals of $1/2$ inch, $1/4$ inch, and so on. As the size of the interval decreases by half, the tick length decreases by one. (See Figure 3.17.)

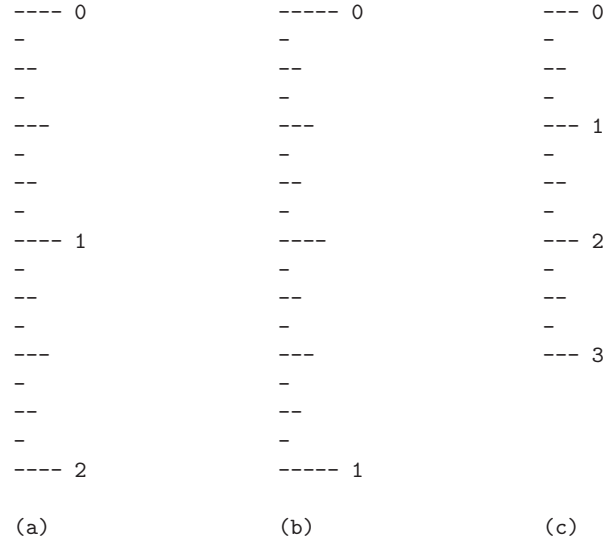


Figure 3.17: Three sample outputs of an English ruler drawing: (a) a 2-inch ruler with major tick length 4; (b) a 1-inch ruler with major tick length 5; (c) a 3-inch ruler with major tick length 3.

Each fraction of an inch also has a numeric label. The longest tick length is called the ***major tick length***. We won't worry about actual distances, however, and just print one tick per line.

A Recursive Approach to Ruler Drawing

Our approach to drawing such a ruler consists of three functions. The main function drawRuler draws the entire ruler. Its arguments are the total number of inches in the ruler, *nInches*, and the major tick length, *majorLength*. The utility function drawOneTick draws a single tick of the given length. It can also be given an optional integer label, which is printed if it is nonnegative.

The interesting work is done by the recursive function drawTicks, which draws the sequence of ticks within some interval. Its only argument is the tick length associated with the interval's central tick. Consider the English ruler with major tick length 5 shown in Figure 3.17(b). Ignoring the lines containing 0 and 1, let us consider how to draw the sequence of ticks lying between these lines. The central tick (at 1/2 inch) has length 4. Observe that the two patterns of ticks above and below this central tick are identical, and each has a central tick of length 3. In general, an interval with a central tick length $L \geq 1$ is composed of the following:

- An interval with a central tick length $L-1$
- A single tick of length L
- An interval with a central tick length $L-1$

With each recursive call, the length decreases by one. When the length drops to zero, we simply return. As a result, this recursive process always terminates. This suggests a recursive process in which the first and last steps are performed by calling the drawTicks$(L-1)$ recursively. The middle step is performed by calling the function drawOneTick(L). This recursive formulation is shown in Code Fragment 3.37. As in the factorial example, the code has a base case (when $L = 0$). In this instance we make two recursive calls to the function.

```
                                          // one tick with optional label
void drawOneTick(int tickLength, int tickLabel = -1) {
  for (int i = 0; i < tickLength; i++)
    cout << "-";
  if (tickLabel >= 0) cout << " " << tickLabel;
  cout << "\n";
}
void drawTicks(int tickLength) {          // draw ticks of given length
  if (tickLength > 0) {                   // stop when length drops to 0
    drawTicks(tickLength-1);              // recursively draw left ticks
    drawOneTick(tickLength);              // draw center tick
    drawTicks(tickLength-1);              // recursively draw right ticks
  }
}
void drawRuler(int nInches, int majorLength) {// draw the entire ruler
  drawOneTick(majorLength, 0);            // draw tick 0 and its label
  for (int i = 1; i <= nInches; i++) {
    drawTicks(majorLength-1);             // draw ticks for this inch
    drawOneTick(majorLength, i);          // draw tick i and its label
  }
}
```

Code Fragment 3.37: A recursive implementation of a function that draws a ruler.

Illustrating Ruler Drawing using a Recursion Trace

The recursive execution of the recursive drawTicks function, defined above, can be visualized using a recursion trace.

The trace for drawTicks is more complicated than in the factorial example, however, because each instance makes two recursive calls. To illustrate this, we show the recursion trace in a form that is reminiscent of an outline for a document. See Figure 3.18.

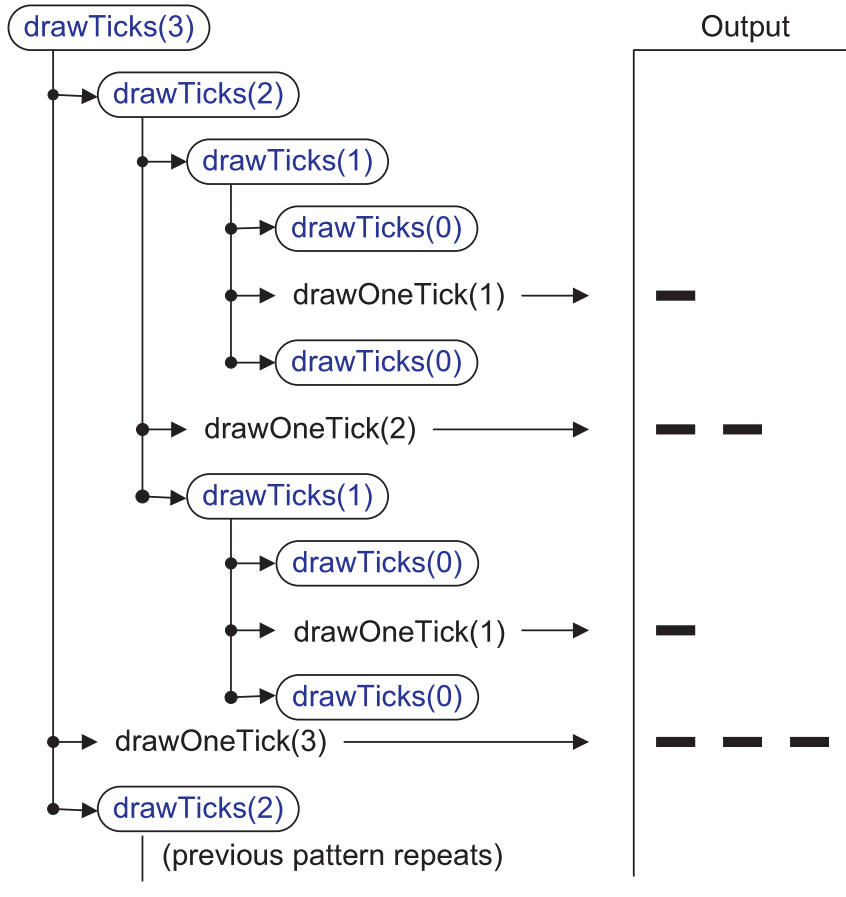


Figure 3.18: A partial recursion trace for the call drawTicks(3). The second pattern of calls for drawTicks(2) is not shown, but it is identical to the first.

Throughout this book, we see many other examples of how recursion can be used in the design of data structures and algorithms.

Further Illustrations of Recursion

As we discussed above, ***recursion*** is the concept of defining a function that makes a call to itself. When a function calls itself, we refer to this as a ***recursive*** call. We also consider a function M to be recursive if it calls another function that ultimately leads to a call back to M.

The main benefit of a recursive approach to algorithm design is that it allows us to take advantage of the repetitive structure present in many problems. By making our algorithm description exploit this repetitive structure in a recursive way, we can often avoid complex case analyses and nested loops. This approach can lead to more readable algorithm descriptions, while still being quite efficient.

In addition, recursion is a useful way for defining objects that have a repeated similar structural form, such as in the following examples.

Example 3.1: *Modern operating systems define file-system directories (which are also sometimes called "folders") in a recursive way. Namely, a file system consists of a top-level directory, and the contents of this directory consists of files and other directories, which in turn can contain files and other directories, and so on. The base directories in the file system contain only files, but by using this recursive definition, the operating system allows for directories to be nested arbitrarily deep (as long as there is enough space in memory).*

Example 3.2: *Much of the syntax in modern programming languages is defined in a recursive way. For example, we can define an argument list in* C++ *using the following notation:*

argument-list:
argument
argument-list , argument

In other words, an argument list consists of either (i) *an argument or* (ii) *an argument list followed by a comma and an argument. That is, an argument list consists of a comma-separated list of arguments. Similarly, arithmetic expressions can be defined recursively in terms of primitives (like variables and constants) and arithmetic expressions.*

Example 3.3: *There are many examples of recursion in art and nature. One of the most classic examples of recursion used in art is in the Russian Matryoshka dolls. Each doll is made of solid wood or is hollow and contains another Matryoshka doll inside it.*

3.5.1 Linear Recursion

The simplest form of recursion is ***linear recursion***, where a function is defined so that it makes at most one recursive call each time it is invoked. This type of recursion is useful when we view an algorithmic problem in terms of a first or last element plus a remaining set that has the same structure as the original set.

Summing the Elements of an Array Recursively

Suppose, for example, we are given an array, A, of n integers that we want to sum together. We can solve this summation problem using linear recursion by observing that the sum of all n integers in A is equal to $A[0]$, if $n = 1$, or the sum of the first $n-1$ integers in A plus the last element in A. In particular, we can solve this summation problem using the recursive algorithm described in Code Fragment 3.38.

Algorithm LinearSum(A,n):
Input: A integer array A and an integer $n \geq 1$, such that A has at least n elements
Output: The sum of the first n integers in A

```
if n = 1 then
  return A[0]
else
  return LinearSum(A, n − 1) + A[n − 1]
```

Code Fragment 3.38: Summing the elements in an array using linear recursion.

This example also illustrates an important property that a recursive function should always possess—the function terminates. We ensure this by writing a nonrecursive statement for the case $n = 1$. In addition, we always perform the recursive call on a smaller value of the parameter $(n-1)$ than that which we are given (n), so that, at some point (at the "bottom" of the recursion), we will perform the nonrecursive part of the computation (returning $A[0]$). In general, an algorithm that uses linear 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 eventually reaches a base case, and the handling of each base case should not use recursion.
- ***Recur.*** After testing for base cases, we then perform a single recursive call. This recursive step may involve a test that decides which of several possible recursive calls to make, but it should ultimately choose to make just one of these calls each time we perform this step. Moreover, we should define each possible recursive call so that it makes progress towards a base case.

Analyzing Recursive Algorithms using Recursion Traces

We can analyze a recursive algorithm by using a visual tool known as a ***recursion trace***. We used recursion traces, for example, to analyze and visualize the recursive factorial function of Section 3.5, and we similarly use recursion traces for the recursive sorting algorithms of Sections 11.1 and 11.2.

To draw a recursion trace, we create a box for each instance of the function and label it with the parameters of the function. Also, we visualize a recursive call by drawing an arrow from the box of the calling function to the box of the called function. For example, we illustrate the recursion trace of the LinearSum algorithm of Code Fragment 3.38 in Figure 3.19. We label each box in this trace with the parameters used to make this call. Each time we make a recursive call, we draw a line to the box representing the recursive call. We can also use this diagram to visualize stepping through the algorithm, since it proceeds by going from the call for n to the call for $n-1$, to the call for $n-2$, and so on, all the way down to the call for 1. When the final call finishes, it returns its value back to the call for 2, which adds in its value, and returns this partial sum to the call for 3, and so on, until the call for $n-1$ returns its partial sum to the call for n.

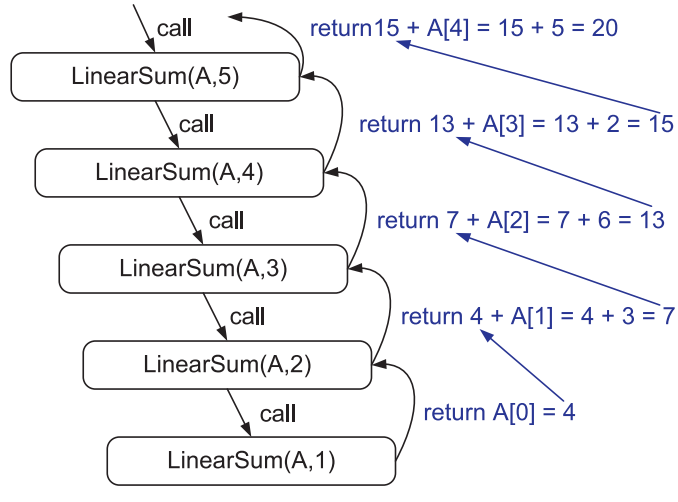


Figure 3.19: Recursion trace for an execution of LinearSum(A,n) with input parameters $A = \{4,3,6,2,5\}$ and $n = 5$.

From Figure 3.19, it should be clear that for an input array of size n, Algorithm LinearSum makes n calls. Hence, it takes an amount of time that is roughly proportional to n, since 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 A) is also roughly proportional to n, since we need a constant amount of memory space for each of the n boxes in the trace at the

time we make the final recursive call (for $n = 1$).

Reversing an Array by Recursion

Next, let us consider the problem of reversing the n elements of an array, A, 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 an array can be achieved by swapping the first and last elements and then recursively reversing the remaining elements in the array. We describe the details of this algorithm in Code Fragment 3.39, using the convention that the first time we call this algorithm we do so as ReverseArray$(A, 0, n-1)$.

Algorithm ReverseArray(A, i, j):
Input: An array A and nonnegative integer indices i and j
Output: The reversal of the elements in A starting at index i and ending at j
if $i < j$ **then**
Swap $A[i]$ and $A[j]$
ReverseArray$(A, i+1, j-1)$
return

Code Fragment 3.39: Reversing the elements of an array using linear recursion.

Note that, in this algorithm, we actually have two base cases, namely, when $i = j$ and when $i > j$. Moreover, in either case, we simply terminate the algorithm, since a sequence with zero elements or one element is trivially equal to its reversal. Furthermore, note that in the recursive step we are guaranteed to make progress towards one of these two base cases. If n is odd, we eventually reach the $i = j$ case, and if n is even, we eventually reach the $i > j$ case. The above argument immediately implies that the recursive algorithm of Code Fragment 3.39 is guaranteed to terminate.

Defining Problems in Ways that Facilitate Recursion

To design a recursive algorithm for a given problem, it is useful to think of the different ways we can subdivide this problem to define problems that have the same general structure as the original problem. This process sometimes means we need to redefine the original problem to facilitate similar-looking subproblems. For example, with the ReverseArray algorithm, we added the parameters i and j so that a recursive call to reverse the inner part of the array A would have the same structure (and same syntax) as the call to reverse all of A. Then, rather than initially calling the algorithm as ReverseArray(A), we call it initially as ReverseArray$(A, 0, n-1)$. In general, 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.

Tail Recursion

Using recursion can often be a useful tool for designing algorithms that have elegant, short definitions. But this usefulness does come at a modest cost. When we use a recursive algorithm to solve a problem, we have to use some of the memory locations in our computer to keep track of the state of each active recursive call. When computer memory is at a premium, then it is useful in some cases to be able to derive nonrecursive algorithms from recursive ones.

We can use the stack data structure, discussed in Section 5.1, to convert a recursive algorithm into a nonrecursive algorithm, but there are some instances when we can do this conversion more easily and efficiently. Specifically, we can easily convert algorithms that use ***tail recursion***. An algorithm uses tail recursion if it uses linear recursion and the algorithm makes a recursive call as its very last operation. For example, the algorithm of Code Fragment 3.39 uses tail recursion to reverse the elements of an array.

It is not enough that the last statement in the function definition includes a recursive call, however. In order for a function to use tail recursion, the recursive call must be absolutely the last thing the function does (unless we are in a base case, of course). For example, the algorithm of Code Fragment 3.38 does not use tail recursion, even though its last statement includes a recursive call. This recursive call is not actually the last thing the function does. After it receives the value returned from the recursive call, it adds this value to $A[n-1]$ and returns this sum. That is, the last thing this algorithm does is an add, not a recursive call.

When an algorithm uses tail recursion, we can convert the recursive algorithm into a nonrecursive one, by iterating through the recursive calls rather than calling them explicitly. We illustrate this type of conversion by revisiting the problem of reversing the elements of an array. In Code Fragment 3.40, we give a nonrecursive algorithm that performs this task by iterating through the recursive calls of the algorithm of Code Fragment 3.39. We initially call this algorithm as $\mathsf{IterativeReverseArray}(A, 0, n-1)$.

Algorithm $\mathsf{IterativeReverseArray}(A, i, j)$:
Input: An array A and nonnegative integer indices i and j
Output: The reversal of the elements in A starting at index i and ending at j

```
while i < j do
    Swap A[i] and A[j]
    i ← i + 1
    j ← j − 1
return
```

Code Fragment 3.40: Reversing the elements of an array using iteration.

3.5.2 Binary Recursion

When an algorithm makes two recursive calls, we say that it uses ***binary recursion***. These calls can, for example, be used to solve two similar halves of some problem, as we did in Section 3.5 for drawing an English ruler. As another application of binary recursion, let us revisit the problem of summing the n elements of an integer array A. In this case, we can sum the elements in A by: (*i*) recursively summing the elements in the first half of A; (*ii*) recursively summing the elements in the second half of A; and (*iii*) adding these two values together. We give the details in the algorithm of Code Fragment 3.41, which we initially call as $\mathsf{BinarySum}(A,0,n)$.

Algorithm $\mathsf{BinarySum}(A,i,n)$:
  Input: An array A and integers i and n
  Output: The sum of the n integers in A starting at index i
  if $n = 1$ **then**
    return $A[i]$
  return $\mathsf{BinarySum}(A,i,\lceil n/2\rceil) + \mathsf{BinarySum}(A,i+\lceil n/2\rceil,\lfloor n/2\rfloor)$

Code Fragment 3.41: Summing the elements in an array using binary recursion.

To analyze Algorithm BinarySum, we consider, for simplicity, the case where n is a power of two. The general case of arbitrary n is considered in Exercise R-4.5. Figure 3.20 shows the recursion trace of an execution of function $\mathsf{BinarySum}(0,8)$. We label each box with the values of parameters i and n, which represent the starting index and length of the sequence of elements to be summed, respectively. Notice that the arrows in the trace go from a box labeled (i,n) to another box labeled $(i,n/2)$ or $(i+n/2,n/2)$. That is, the value of parameter n is halved at each recursive call. Thus, the depth of the recursion, that is, the maximum number of function instances that are active at the same time, is $1+\log_2 n$. Thus, Algorithm BinarySum uses an amount of additional space roughly proportional to this value. This is a big improvement over the space needed by the LinearSum function of Code Fragment 3.38. The running time of Algorithm BinarySum is still roughly proportional to n, however, since each box is visited in constant time when stepping through our algorithm and there are $2n-1$ boxes.

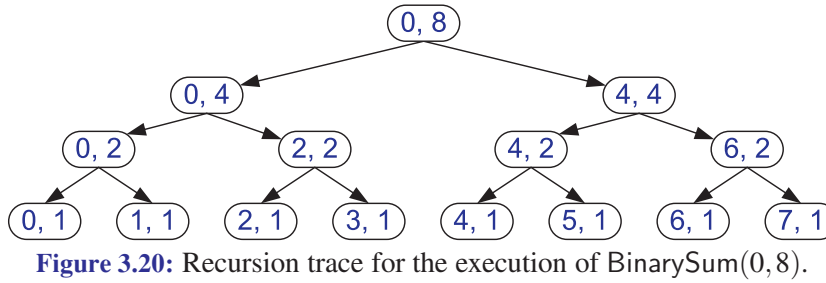


Figure 3.20: Recursion trace for the execution of $\mathsf{BinarySum}(0,8)$.

Computing Fibonacci Numbers via Binary Recursion

Let us consider the problem of computing the kth Fibonacci number. Recall from Section 2.2.3, that the Fibonacci numbers are recursively defined as follows:

$$\begin{aligned} F_0 &= 0 \\ F_1 &= 1 \\ F_i &= F_{i-1} + F_{i-2} \quad \text{for } i > 1 \end{aligned}$$

By directly applying this definition, Algorithm BinaryFib, shown in Code Fragment 3.42, computes the sequence of Fibonacci numbers using binary recursion.

Algorithm BinaryFib(k):
Input: Nonnegative integer k
Output: The kth Fibonacci number F_k
if $k \leq 1$ **then**
  return k
else
  return BinaryFib($k-1$) + BinaryFib($k-2$)

Code Fragment 3.42: Computing the kth Fibonacci number using binary recursion.

Unfortunately, in spite of the Fibonacci definition looking like a binary recursion, using this technique is inefficient in this case. In fact, it takes an exponential number of calls to compute the kth Fibonacci number in this way. Specifically, let n_k denote the number of calls performed in the execution of BinaryFib(k). Then, we have the following values for the n_k's:

$$\begin{aligned} n_0 &= 1 \\ n_1 &= 1 \\ n_2 &= n_1 + n_0 + 1 = 1 + 1 + 1 = 3 \\ n_3 &= n_2 + n_1 + 1 = 3 + 1 + 1 = 5 \\ n_4 &= n_3 + n_2 + 1 = 5 + 3 + 1 = 9 \\ n_5 &= n_4 + n_3 + 1 = 9 + 5 + 1 = 15 \\ n_6 &= n_5 + n_4 + 1 = 15 + 9 + 1 = 25 \\ n_7 &= n_6 + n_5 + 1 = 25 + 15 + 1 = 41 \\ n_8 &= n_7 + n_6 + 1 = 41 + 25 + 1 = 67 \end{aligned}$$

If we follow the pattern forward, we see that the number of calls more than doubles for each two consecutive indices. That is, n_4 is more than twice n_2, n_5 is more than twice n_3, n_6 is more than twice n_4, and so on. Thus, $n_k > 2^{k/2}$, which means that BinaryFib(k) makes a number of calls that are exponential in k. In other words, using binary recursion to compute Fibonacci numbers is very inefficient.

Computing Fibonacci Numbers via Linear Recursion

The main problem with the approach above, based on binary recursion, is that the computation of Fibonacci numbers is really a linearly recursive problem. It is not a good candidate for using binary recursion. We simply got tempted into using binary recursion because of the way the kth Fibonacci number, F_k, depends on the two previous values, F_{k-1} and F_{k-2}. But we can compute F_k much more efficiently using linear recursion.

In order to use linear recursion, however, we need to slightly redefine the problem. One way to accomplish this conversion is to define a recursive function that computes a pair of consecutive Fibonacci numbers (F_k, F_{k-1}) using the convention $F_{-1} = 0$. Then we can use the linearly recursive algorithm shown in Code Fragment 3.43.

Algorithm LinearFibonacci(k):
Input: A nonnegative integer k
Output: Pair of Fibonacci numbers (F_k, F_{k-1})
if $k \leq 1$ **then**
  return $(k, 0)$
else
  $(i, j) \leftarrow$ LinearFibonacci($k-1$)
  return $(i+j, i)$

Code Fragment 3.43: Computing the kth Fibonacci number using linear recursion.

The algorithm given in Code Fragment 3.43 shows that using linear recursion to compute Fibonacci numbers is much more efficient than using binary recursion. Since each recursive call to LinearFibonacci decreases the argument k by 1, the original call LinearFibonacci(k) results in a series of $k-1$ additional calls. That is, computing the kth Fibonacci number via linear recursion requires k function calls. This performance is significantly faster than the exponential time needed by the algorithm based on binary recursion, which was given in Code Fragment 3.42. Therefore, when using binary recursion, we should first try to fully partition the problem in two (as we did for summing the elements of an array) or we should be sure that overlapping recursive calls are really necessary.

Usually, we can eliminate overlapping recursive calls by using more memory to keep track of previous values. In fact, this approach is a central part of a technique called ***dynamic programming***, which is related to recursion and is discussed in Section 12.2.

3.5.3 Multiple Recursion

Generalizing from binary recursion, we use ***multiple recursion*** when a function may make multiple recursive calls, with that number potentially being more than two. One of the most common applications of this type of recursion is used when we want to enumerate various configurations in order to solve a combinatorial puzzle. For example, the following are all instances of ***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 though the feasible configurations left, 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. In addition, such an algorithm can use multiple recursion to work through the configurations in a systematic way. We show pseudo-code for such an algorithm in Code Fragment 3.44. To keep the description general enough to be used with other puzzles, the algorithm enumerates and tests all k-length sequences without repetitions of the elements of a given set U. We build the sequences of k elements by 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 the 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 3.44 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**
  Remove e from U {e is now being used}
  Add e to the end of S
  if $k = 1$ **then**
    Test whether S is a configuration that solves the puzzle
    if S solves the puzzle **then**
      return "Solution found: " S
  else
    PuzzleSolve$(k-1,S,U)$
  Add e back to U {e is now unused}
  Remove e from the end of S

Code Fragment 3.44: Solving a combinatorial puzzle by enumerating and testing all possible configurations.

In Figure 3.21, 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 3.21.

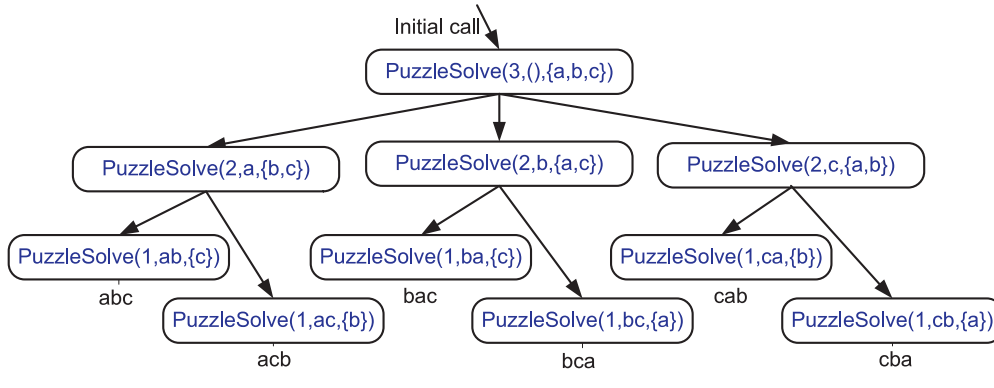


Figure 3.21: Recursion trace for an execution of PuzzleSolve$(3,S,U)$, where S is empty and $U = \{a,b,c\}$. This execution generates and tests all permutations of a, b, and c. We show the permutations generated directly below their respective boxes.

3.6 Exercises

For help with exercises, please visit the web site, www.wiley.com/college/goodrich.

Reinforcement

R-3.1 Modify the implementation of class Scores so that at most $\lceil maxEnt/2 \rceil$ of the scores can come from any one single player.

R-3.2 Suppose that two entries of an array A are equal to each other. After running the insertion-sort algorithm of Code Fragment 3.7, will they appear in the same relative order in the final sorted order or in reverse order? Explain your answer.

R-3.3 Give a C++ code fragment that, given a $n \times n$ matrix M of type **float**, replaces M with its transpose. Try to do this without the use of a temporary matrix.

R-3.4 Describe a way to use recursion to compute the sum of all the elements in a $n \times n$ (two-dimensional) array of integers.

R-3.5 Give a recursive definition of a singly linked list.

R-3.6 Add a function size() to our C++ implementation of a singly link list. Can you design this function so that it runs in $O(1)$ time?

R-3.7 Give an algorithm for finding the penultimate (second to last) node in a singly linked list where the last element is indicated by a null next link.

R-3.8 Give a fully generic implementation of the doubly linked list data structure of Section 3.3.3 by using a templated class.

R-3.9 Give a more robust implementation of the doubly linked list data structure of Section 3.3.3, which throws an appropriate exception if an illegal operation is attempted.

R-3.10 Describe a nonrecursive function for finding, by link hopping, the middle node of a doubly linked list with header and trailer sentinels. (Note: This function must only use link hopping; it cannot use a counter.) What is the running time of this function?

R-3.11 Describe a recursive algorithm for finding the maximum element in an array A of n elements. What is your running time and space usage?

R-3.12 Draw the recursion trace for the execution of function ReverseArray$(A,0,4)$ (Code Fragment 3.39) on array $A = \{4,3,6,2,5\}$.

R-3.13 Draw the recursion trace for the execution of function PuzzleSolve$(3,S,U)$ (Code Fragment 3.44), where S is empty and $U = \{a,b,c,d\}$.

R-3.14 Write a short C++ function that repeatedly selects and removes a random entry from an n-element array until the array holds no more entries. Assume that you have access to a function random(k), which returns a random integer in the range from 0 to k.

R-3.15 Give a fully generic implementation of the circularly linked list data structure of Section 3.4.1 by using a templated class.

R-3.16 Give a more robust implementation of the circularly linked list data structure of Section 3.4.1, which throws an appropriate exception if an illegal operation is attempted.

R-3.17 Write a short C++ function to count the number of nodes in a circularly linked list.

Creativity

C-3.1 In the Tic-Tac-Toe example, we used 1 for player X and -1 for player O. Explain how to modify the program's counting trick to decide the winner if we had used 1 for player X and 4 for player O instead. Could we use any combination of values a and b for the two players? Explain.

C-3.2 Give C++ code for performing add(e) and remove(i) functions for game entries stored in an array a, as in class Scores in Section 3.1.1, except this time, don't maintain the game entries in order. Assume that we still need to keep n entries stored in indices 0 to $n-1$. Try to implement the add and remove functions without using any loops, so that the number of steps they perform does not depend on n.

C-3.3 Let A be an array of size $n \geq 2$ containing integers from 1 to $n-1$, inclusive, with exactly one repeated. Describe a fast algorithm for finding the integer in A that is repeated.

C-3.4 Let B be an array of size $n \geq 6$ containing integers from 1 to $n-5$, inclusive, with exactly five repeated. Describe a good algorithm for finding the five integers in B that are repeated.

C-3.5 Suppose you are designing a multi-player game that has $n \geq 1000$ players, numbered 1 to n, interacting in an enchanted forest. The winner of this game is the first player who can meet all the other players at least once (ties are allowed). Assuming that there is a function meet(i, j), which is called each time a player i meets a player j (with $i \neq j$), describe a way to keep track of the pairs of meeting players and who is the winner.

C-3.6 Give a recursive algorithm to compute the product of two positive integers, m and n, using only addition and subtraction.

C-3.7 Describe a fast recursive algorithm for reversing a singly linked list L, so that the ordering of the nodes becomes opposite of what it was before.

C-3.8 Describe a good algorithm for concatenating two singly linked lists L and M, with header sentinels, into a single list L' that contains all the nodes of L followed by all the nodes of M.

C-3.9 Give a fast algorithm for concatenating two doubly linked lists L and M, with header and trailer sentinel nodes, into a single list L'.

C-3.10 Describe in detail how to swap two nodes x and y (and not just their contents) in a singly linked list L given references only to x and y. Repeat this exercise for the case when L is a doubly linked list. Which algorithm takes more time?

C-3.11 Describe in detail an algorithm for reversing a singly linked list L using only a constant amount of additional space and not using any recursion.

C-3.12 In the ***Towers of Hanoi*** puzzle, we are given a platform with three pegs, a, b, and c, sticking out of it. On peg a is a stack of n disks, each larger than the next, so that the smallest is on the top and the largest is on the bottom. The puzzle is to move all the disks from peg a to peg c, moving one disk at a time, so that we never place a larger disk on top of a smaller one. Describe a recursive algorithm for solving the Towers of Hanoi puzzle for arbitrary n.

(Hint: Consider first the subproblem of moving all but the nth disk from peg a to another peg using the third as "temporary storage.")

C-3.13 Describe a recursive function for converting a string of digits into the integer it represents. For example, `"13531"` represents the integer $13,531$.

C-3.14 Describe a recursive algorithm that counts the number of nodes in a singly linked list.

C-3.15 Write a recursive C++ program that will output all the subsets of a set of n elements (without repeating any subsets).

C-3.16 Write a short recursive C++ function that finds the minimum and maximum values in an array of **int** values without using any loops.

C-3.17 Describe a recursive algorithm that will check if an array A of integers contains an integer $A[i]$ that is the sum of two integers that appear earlier in A, that is, such that $A[i] = A[j] + A[k]$ for $j, k < i$.

C-3.18 Write a short recursive C++ function that will rearrange an array of **int** values so that all the even values appear before all the odd values.

C-3.19 Write a short recursive C++ function that takes a character string s and outputs its reverse. So for example, the reverse of `"pots&pans"` would be `"snap&stop"`.

C-3.20 Write a short recursive C++ function that determines if a string s is a palindrome, that is, it is equal to its reverse. For example, `"racecar"` and `"gohangasalamiimalasagnahog"` are palindromes.

C-3.21 Use recursion to write a C++ function for determining if a string s has more vowels than consonants.

C-3.22 Suppose you are given two circularly linked lists, L and M, that is, two lists of nodes such that each node has a nonnull next node. Describe a fast algorithm for telling if L and M are really the same list of nodes but with different (cursor) starting points.

C-3.23 Given a circularly linked list L containing an even number of nodes, describe how to split L into two circularly linked lists of half the size.

Projects

P-3.1 Write a C++ function that takes two three-dimensional integer arrays and adds them componentwise.

P-3.2 Write a C++ program for a matrix class that can add and multiply arbitrary two-dimensional arrays of integers. Do this by overloading the addition ("+") and multiplication ("*") operators.

P-3.3 Write a class that maintains the top 10 scores for a game application, implementing the add and remove functions of Section 3.1.1, but use a singly linked list instead of an array.

P-3.4 Perform the previous project but use a doubly linked list. Moreover, your implementation of remove(i) should make the fewest number of pointer hops to get to the game entry at index i.

P-3.5 Perform the previous project but use a linked list that is both circularly linked and doubly linked.

P-3.6 Write a program for solving summation puzzles by enumerating and testing all possible configurations. Using your program, solve the three puzzles given in Section 3.5.3.

P-3.7 Write a program that can perform encryption and decryption using an arbitrary substitution cipher. In this case, the encryption array is a random shuffling of the letters in the alphabet. Your program should generate a random encryption array, its corresponding decryption array, and use these to encode and decode a message.

P-3.8 Write a program that can solve instances of the Tower of Hanoi problem (from Exercise C-3.12).

Chapter Notes

The fundamental data structures of arrays and linked lists, as well as recursion, discussed in this chapter, belong to the folklore of computer science. They were first chronicled in the computer science literature by Knuth in his seminal book on *Fundamental Algorithms* [59].