DATA STRUCTURES

A Pseudocode Approach with C

Richard F. Gilberg & Behrouz A. Forouzan

SECOND EDITION

Data Structures: A Pseudocode Approach with C, Second Edition
Richard F. Gilberg
Behrouz A. Forouzan

Senior Product Manager:
Alyssa Pratt

Senior Acquisitions Editor:
Amy Yarnevich

Production Editor:
BobbiJo Frasca

Senior Marketing Manager:
Karen Seitz

Associate Product Manager:
Mirella Misiaszek

Editorial Assistant:
Amanda Piantedosi

Senior Manufacturing Coordinator:
Trevor Kallop

Cover Designer:
Abby Scholz

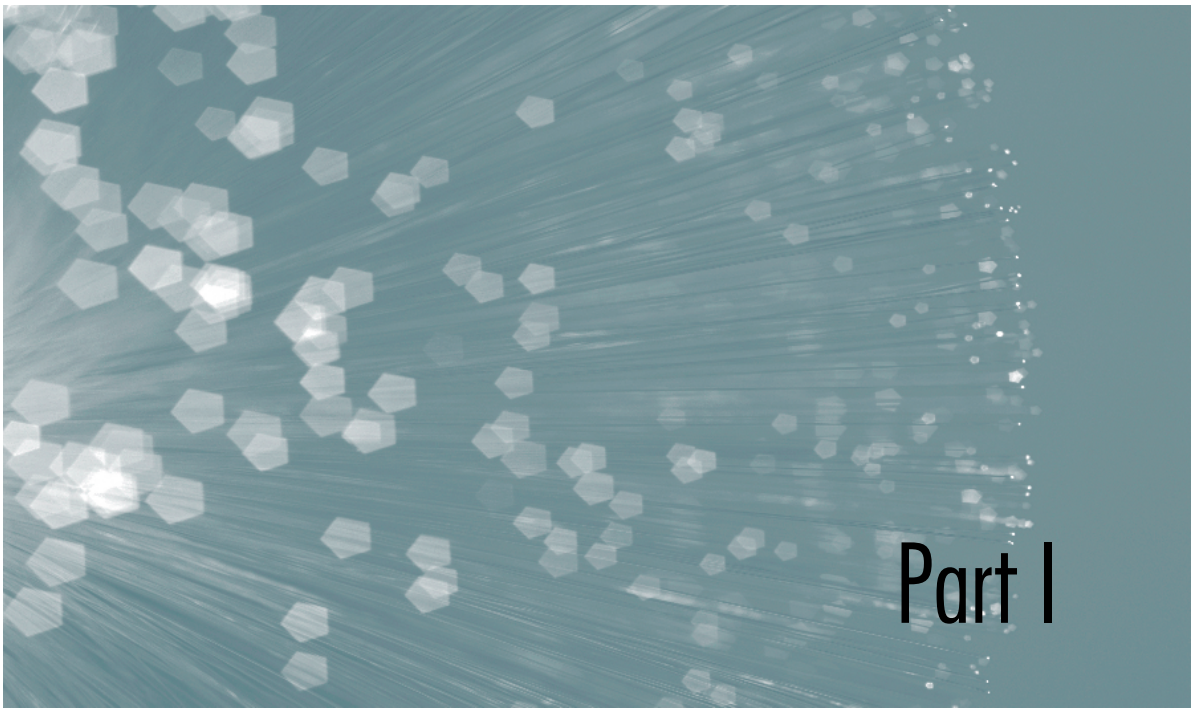
Printed in the United States of America

1 2 3 4 5 6 7 8 9 QWT 09 08 07 06 05

For more information, contact Course Technology, 25 Thomson Place, Boston, Massachusetts, 02210.

Or find us on the World Wide Web at: www.course.com

ISBN-13: 978-0-534-39080-8
ISBN-10: 0-534-39080-3



Part I

Introduction

There are several concepts that are essential to an understanding of this text. We discuss these concepts in the first two chapters. Chapter 1 "Basic Concepts," covers general materials that we use throughout the book. Chapter 2 "Recursion," discusses the concept of recursion. Figure I-1 shows the organization of Part I.

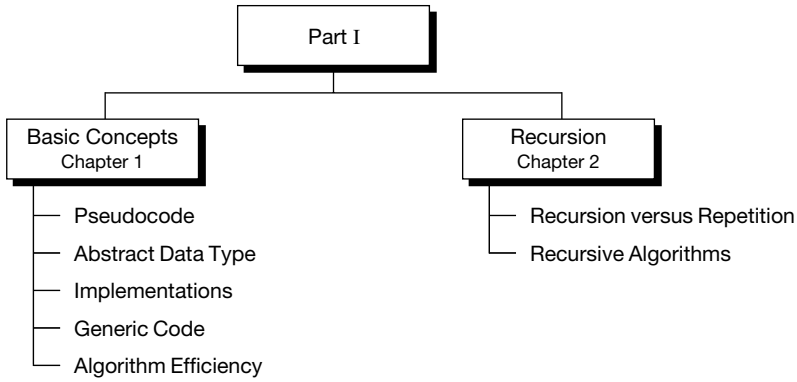


FIGURE I-1 Part I Contents

Chapters Covered

This part includes two chapters.

Chapter 1: Basic Concepts

The first chapter covers concepts that we use throughout the text. You may find that some of them are a review of material from your programming course. The major concepts are outlined below.

Pseudocode

In all of our texts, we use the basic tenet, "Design comes before code." Throughout the text, therefore, we separate algorithm design from the code that implements it in a specific language. Although the underlying language in this book is C, pseudocode allows us to separate the algorithm from the implementation.

Abstract Data Type

An abstract data type (ADT) implements a set of algorithms generically so that they can be applied to any data type or construct. The beauty of an ADT implementation is that the algorithms can handle any data type whether it is a simple integer or a complex record.

ADT Implementations

In general, there are two basic data structures that can be used to implement an abstract data type: arrays and linked lists. We discuss basic linked list concepts in Chapter 1 and expand on them as necessary in subsequent chapters.

Generic Code for ADTs

To implement the ADT concept, we need to use generic code. Each language provides a different set of tools to implement generic code. The C language uses two tools: pointer to *void* and pointer to function.

Algorithm Efficiency

While many authors argue that today's computers and compilers make algorithm efficiency an academic discussion, we believe that an understanding of algorithm efficiency provides the framework for writing better algorithms. Although we discuss the efficiency of specific algorithms when we develop them, in this chapter we discuss the basic concepts and tools for discussing algorithm efficiency.

Chapter 2: Recursion

In Chapter 2 we discuss recursion, a concept that is often skipped in an introductory programming course. We need to understand recursion to discuss data structures because many of the abstract data types are recursive by nature and algorithms that handle them can be better understood using recursion. We use recursive algorithms extensively, especially in Part III, "Non-Linear Lists."

Recursion versus Repetition

The first part of the chapter compares and contrasts recursion and repetition and when each should be used.

Recursive Algorithms

Although recursive algorithms are generally elegant, they can be difficult to understand. In the second part of the chapter, we introduce several algorithms to make the recursive concept clear and to provide a sense of design for creating recursive algorithms.

Chapter 1
Basic Concepts

This text assumes that the student has a solid foundation in structured programming principles and has written programs of moderate complexity. Although the text uses C for all of its implementation examples, the design and logic of the data structure algorithms are based on pseudocode. This approach creates a language-independent environment for the algorithms.

In this chapter we establish a background for the tools used in the rest of the text, most specifically pseudocode, the abstract data type, algorithm efficiency analysis, and the concepts necessary to create generic code.

1.1 Pseudocode

Although several tools are used to define algorithms, one of the most common is **pseudocode**. Pseudocode is an English-like representation of the algorithm logic. It is part English, part structured code. The English part provides a relaxed syntax that describes *what* must be done without showing unnecessary details such as error messages. The code part consists of an extended version of the basic algorithmic constructs—sequence, selection, and iteration.

> One of the most common tools for defining algorithms is pseudocode, which is part English, part structured code.

In this text we use pseudocode to represent both data structures and code. Data items do not need to be declared. The first time we use a data name in an algorithm, it is automatically declared. Its type is determined by context. The following statement declares a numeric data item named `count` and sets its value to zero.

```
set count to 0
```

The structure of the data, on the other hand, must be declared. We use a simple syntactical statement that begins with a structure name and concludes with the keyword *end* and the name of the structure. Within the structure we list the structural elements by indenting the data items as shown below.

```
node
       data
       link
end node
```

This data definition describes a node in a self-referential list that consists of a nested structure (`data`) and a pointer to the next node (link). An element's type is implied by its name and usage in the algorithm.

As mentioned, pseudocode is used to describe an algorithm. To facilitate a discussion of the algorithm statements, we number them using the hierarchical system shown in Algorithm 1-1. The following sections describe the components of an algorithm. Colored comments provide documentation or clarification when required.

ALGORITHM 1-1 Example of Pseudocode

```
Algorithm sample (pageNumber)
This algorithm reads a file and prints a report.
   Pre    pageNumber passed by reference
   Post   Report Printed
          pageNumber contains number of pages in report
   Return Number of lines printed
 1 loop (not end of file)
   1   read file
   2   if (full page)
       1   increment page number
       2   write page heading
   3   end if
   4   write report line
   5   increment line count
 2 end loop
 3 return line count
end sample
```

Algorithm Header

Each algorithm begins with a header that names it, lists its parameters, and describes any preconditions and postconditions. This information is important because it serves to document the algorithm. Therefore, the header information must be complete enough to communicate to the programmer everything he or she must know to write the algorithm. In Algorithm 1-1 there is only one parameter, the page number.

Purpose, Conditions, and Return

The purpose is a short statement about what the algorithm does. It needs to describe only the general algorithm processing. It should not attempt to describe all of the processing. For example, in Algorithm 1-1 the purpose does not need to state that the file will be opened or how the report will be printed. Similarly, in the search example the purpose does not need to state which of the possible array searches will be used.

The precondition lists any precursor requirements for the parameters. For example, in Algorithm 1-1 the algorithm that calls `sample` must pass the page number by reference. Sometimes there are no preconditions, in which case we still list the precondition with a statement that nothing is required, as shown below.

```
Pre    nothing
```

If there are several input parameters, the precondition should be shown for each. For example, a simple array search algorithm has the following header:

```
Algorithm search  (list, argument, location)
Search array for specific item and return index location.
   Pre    list contains data array to be searched
          argument contains data to be located in list
   Post   location contains matching index
          -or- undetermined if not found
   Return true if found, false if not found
```


In `search` the precondition specifies that the two input parameters, list and argument, must be initialized. If a binary search were being used, the precondition would also state that the array data must be sorted.

The postcondition identifies any action taken and the status of any output parameters. In Algorithm 1-1 the postcondition contains two parts. First, it states that the report has been printed. Second, the reference parameter, `pageNumber`, contains the updated number of pages in the report. In the search algorithm shown above, there is only one postcondition, which may be one of two different values.

If a value is returned, it is identified by a **return condition**. Often there is none, and no return condition is needed. In Algorithm 1-1 we return the number of lines printed. The search algorithm returns true if the argument was found, false if it was not found.

Statement Numbers

Statements are numbered using an abbreviated decimal notation in which only the last of the number sequence is shown on each statement. The expanded number of the statement in Algorithm 1-1 that reads the file is 1.1.

The statement that writes the page heading is 1.2.2. This technique allows us to identify an individual statement while providing statements that are easily read.

Variables

To ensure that the meaning is understood, we use **intelligent data names**—that is, names that describe the meaning of the data. However, it is not necessary to define the variables used in an algorithm, especially when the name indicates the context of the data.

The selection of the name for an algorithm or variable goes a long way toward making the algorithm and its coded implementation more readable. In general, you should follow these rules:

1. Do not use single-character names.
2. Do not use generic names in application programs. Examples of generic names are `count`, `sum`, `total`, `row`, `column`, and `file`. In a program of any size there are several counts, sums, and totals. Rather, add an intelligent qualifier to the generic name so that the reader knows exactly to which piece of data the name refers. For example, `studentCount` and `numberOfStudents` are both better than `count`.
3. Abbreviations are not excluded as intelligent data names. For example, `stuCnt` is a good abbreviation for `student count`, and `numOfStu` is a good abbreviation for `number of students`. Note, however, that `noStu` would not be a good abbreviation for `number of students` because it is too easily read as *no students*.

Statement Constructs

When he first proposed the structured programming model, Edsger Dijkstra stated that any algorithm could be written using only three programming **constructs**: sequence, selection, and loop. Our pseudocode contains only these three basic constructs. The implementation of these constructs relies on the richness of the implementation language. For example, the loop can be implemented as a *while, do…while,* or *for* statement in the C language.

Sequence

A **sequence** is one or more statements that do not alter the execution path *within an algorithm*. Although it is obvious that statements such as assign and add are sequence statements, it is not so obvious that a call to other algorithms is also considered a sequence statement. The reason calls are considered sequential statements lies in the structured programming concept that each algorithm has only one entry and one exit. Furthermore, when an algorithm completes, it returns to the statement immediately after the call that invoked it. Therefore, we can consider an algorithm call a sequence statement. In Algorithm 1-1 statements 1.2.1 and 1.2.2 are sequence statements.

Selection

A **selection statement** evaluates a condition and executes zero or more alternatives. The results of the evaluation determine which alternates are taken.

The typical selection statement is the two-way selection as implemented in an *if* statement. Whereas most languages provide for multiway selections, such as the *switch* in C, we provide none in the pseudocode. The parts of the selection are identified by indentation, as shown in the short pseudocode statement below.

```
1 if (condition)
  1    action1
2 else
  1    action2
3 end if
```

Statement 1.2 in Algorithm 1-1 is an example of a selection statement. The end of the selection is indicated by the *end if* in statement 1.3.

Loop

A **loop** statement iterates a block of code. The loop that we use in our pseudocode closely resembles the *while* loop. It is a pretest loop; that is, the condition is evaluated before the body of the loop is executed. If the condition is true, the body is executed. If the condition is false, the loop terminates.

In Algorithm 1-1 statement 1 is an example of a loop. The end of the loop is indicated by *end loop* in statement 2.

Algorithm Analysis

For selected algorithms, we follow the algorithm with an analysis section that explains some of its salient points. Not every line of code is explained. Rather, the analysis examines only those points that either need to be emphasized or that may require some clarification. The algorithm analysis also often introduces style or efficiency considerations.

Pseudocode Example

As another example of pseudocode, consider the logic required to calculate the deviation from a mean. In this problem we must first read a series of numbers and calculate their average. Then we subtract the mean from each number and print the number and its deviation. At the end of the calculation, we also print the totals and the average.

The obvious solution is to place the data in an array as they are read. Algorithm 1-2 contains the code for this simple problem as it would be implemented in a callable algorithm.

ALGORITHM 1-2 Print Deviation from Mean for Series

```
Algorithm deviation
   Pre    nothing
   Post   average and numbers with their deviation printed
1  loop (not end of file)
   1   read number into array
   2   add number to total
   3   increment count
2  end loop
3  set average to total / count
4  print average
5  loop (not end of array)
   1   set devFromAve to array element - average
   2   print array element and devFromAve
6  end loop
end deviation
```

Algorithm 1-2 Analysis There are two points worth mentioning in Algorithm 1-2. First, there are no parameters. Second, as previously explained, we do not declare variables. A variable's type and purpose should be easily determined by its name and usage.

1.2 The Abstract Data Type

In the history of programming concepts, we started with nonstructured, linear programs, known as **spaghetti code**, in which the logic flow wound through the program like spaghetti on a plate. Next came the concept of **modular programming**, in which programs were organized in functions, each of which still used a linear coding technique. In the 1970s, the basic principles of **structured programming** were formulated by computer scientists such as Edsger Dijkstra and Niklaus Wirth. They are still valid today.

Atomic and Composite Data

Atomic data are data that consist of a single piece of information; that is, they cannot be divided into other meaningful pieces of data. For example, the integer 4562 may be considered a single integer value. Of course, we can decompose it into digits, but the decomposed digits do not have the same characteristics of the original integer; they are four single-digit integers ranging from 0 to 9. In some languages atomic data are known as scalar data because of their numeric properties.

The opposite of atomic data is **composite data**. Composite data can be broken out into subfields that have meaning. As an example of a composite data item, consider your telephone number. A telephone number actually has three different parts. First, there is the area code. Then, what you consider to be your phone number is actually two different data items, a prefix consisting of a three-digit exchange and the number within the exchange,

consisting of four digits. In the past, these prefixes were names such as DAvenport and CYpress.

Data Type

A **data type** consists of two parts: a set of data and the operations that can be performed on the data. Thus we see that the integer type consists of values (whole numbers in some defined range) and operations (add, subtract, multiply, divide, and any other operations appropriate for the data).

Data Type
1. A set of values 2. A set of operations on values

Table 1-1 shows three data types found in all systems.

Type	Values	Operations
integer	-∞, … , -2, -1, 0, 1, 2,…, ∞	*, +, -, %, /, ++, --, …
floating point	-∞, … , 0.0, … , ∞	*, +, -, /, …
character	\0, …, 'A', 'B', … , 'a', 'b', … , ~	<, >, …

TABLE 1-1 Three Data Types

Data Structure

A **data structure** is an aggregation of atomic and composite data into a set with defined relationships. In this definition *structure* means a set of rules that holds the data together. In other words, if we take a combination of data and fit them into a structure such that we can define its relating rules, we have made a data structure. Data structures can be nested. We can have a data structure that consists of other data structures. For example, we can define the two structures array and record as shown in Table 1-2.

Array	Record
Homogeneous sequence of data or data types known as elements	Heterogeneous combination of data into a single structure with an identified key
Position association among the elements	No association

TABLE 1-2 Data Structure Examples

Most of the programming languages support several data structures. In addition, modern programming languages allow programmers to create new data structures for an application.

Data Structure

1. A combination of elements in which each is either a data type or another data structure
2. A set of associations or relationships (structure) involving the combined elements

Abstract Data Type

Generally speaking, programmers' capabilities are determined by the tools in their tool kits. These tools are acquired by education and experience. A knowledge of data structures is one of those tools.

When we first started programming, there were no abstract data types. If we wanted to read a file, we wrote the code to read the physical file device. It did not take long to realize that we were writing the same code over and over again. So we created what is known today as an **abstract data type (ADT)**. We wrote the code to read a file and placed it in a library for all programmers to use.

This concept is found in modern languages today. The code to read the keyboard is an ADT. It has a data structure, a character, and a set of operations that can be used to read that data structure. Using the ADT we can not only read characters but we can also convert them into different data structures such as integers and strings.

With an ADT users are not concerned with *how* the task is done but rather with *what* it can do. In other words, the ADT consists of a set of definitions that allow programmers to use the functions while hiding the implementation. This generalization of operations with unspecified implementations is known as abstraction. We abstract the essence of the process and leave the implementation details hidden.

The concept of abstraction means:

1. We know *what* a data type can do.
2. *How* it is done is hidden.

Consider the concept of a list. At least four data structures can support a list. We can use a matrix, a linear list, a tree, or a graph. If we place our list in an ADT, users should not be aware of the structure we use. As long as they can insert and retrieve data, it should make no difference how we store the data. Figure 1-1 shows four logical structures that might be used to hold a list.

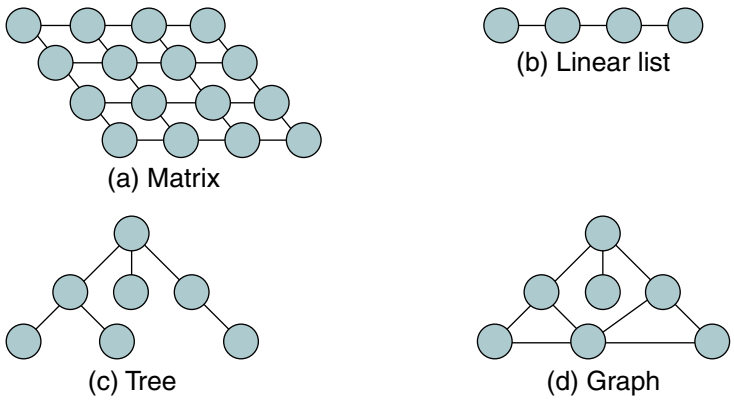


FIGURE 1-1 Some Data Structures

As another example, consider the system analyst who needs to simulate the waiting line of a bank to determine how many tellers are needed to serve customers efficiently. This analysis requires the simulation of a queue. However, queues are not generally available in programming languages. Even if a queue type were available, our analyst would still need some basic queue operations, such as enqueuing (insertion) and dequeuing (deleting), for the simulation.

There are two potential solutions to this problem: (1) we can write a program that simulates the queue our analyst needs (in this case, our solution is good only for the one application at hand) or (2) we can write a queue ADT that can be used to solve any queue problem. If we choose the latter course, our analyst still needs to write a program to simulate the banking application, but doing so is much easier and faster because he or she can concentrate on the application rather than the queue.

We are now ready to define ADT. An abstract data type is a data declaration packaged together with the operations that are meaningful for the data type. In other words, we **encapsulate** the data and the operations on the data, and then we hide them from the user.

Abstract Data Type

1. Declaration of data
2. Declaration of operations
3. Encapsulation of data and operations

We cannot overemphasize the importance of hiding the implementation. The user should not have to know the data structure to use the ADT. Referring to our queue example, the application program should have no knowledge of the data structure. All references to and manipulation of the data in the queue must be handled through defined interfaces to the

structure. Allowing the application program to directly reference the data structure is a common fault in many implementations. This keeps the ADT from being fully portable to other applications.

1.3 Model for an Abstract Data Type

The ADT model is shown in Figure 1-2. The colored area with an irregular outline represents the ADT. Inside the ADT are two different aspects of the model: data structures and functions (public and private). Both are entirely contained in the model and are not within the application program scope. However, the data structures are available to all of the ADT's functions as needed, and a function may call on other functions to accomplish its task. In other words, the data structures and the functions are within scope of each other.

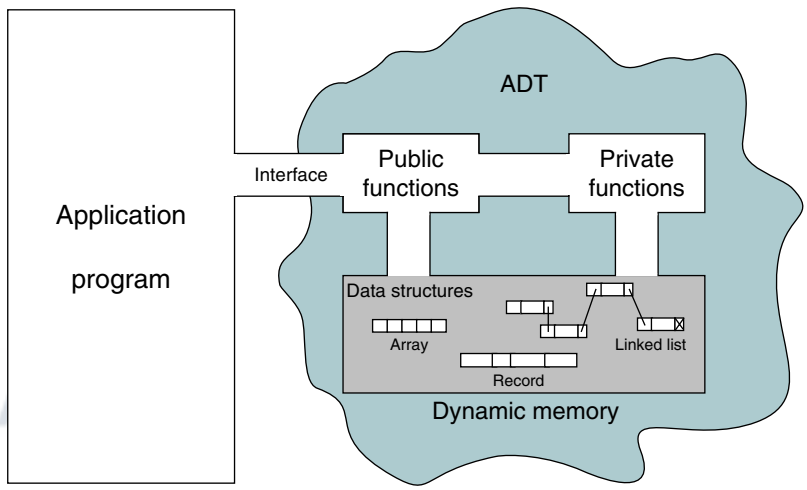


FIGURE 1-2 Abstract Data Type Model

ADT Operations

Data are entered, accessed, modified, and deleted through the external interface drawn as a passageway partially in and partially out of the ADT. Only the public functions are accessible through this interface. For each ADT operation there is an algorithm that performs its specific task. Only the operation name and its parameters are available to the application, and they provide the only interface to the ADT.

ADT Data Structure

When a list is controlled entirely by the program, it is often implemented using simple structures similar to those used in your programming class. Because the abstract data type must hide the implementation from the user, however, all data about the structure must be maintained inside the ADT. Just encapsulating the structure in an ADT is not sufficient. It is also necessary for multiple versions of the structure to be able to coexist. Consequently, we must hide the implementation from the user while being able to store different data.

In this text, we develop ADTs for stacks, queues, lists, binary search trees, AVL trees, B-trees, heaps, and graphs. If you would like a preview, look at the stack ADT in Chapter 3.

1.4 ADT Implementations

There are two basic structures we can use to implement an ADT list: arrays and linked lists.

Array Implementations

In an array, the sequentiality of a list is maintained by the order structure of elements in the array (indexes). Although searching an array for an individual element can be very efficient, addition and deletion of elements are complex and inefficient processes. For this reason arrays are seldom used, especially when the list changes frequently. In addition, array implementations of non-linear lists can become excessively large, especially when there are several successors for each element. Appendix F provides array implementations for two ADTs.

Linked List Implementations

A **linked list** is an ordered collection of data in which each element contains the location of the next element or elements. In a linked list, each element contains two parts: **data** and one or more **links**. The data part holds the application data—the data to be processed. Links are used to chain the data together. They contain pointers that identify the next element or elements in the list.

We can use a linked list to create linear and non-linear structures. In linear linked lists, each element has only zero or one successor. In non-linear linked lists, each element can have zero, one, or more successors.

The major advantage of the linked list over the array is that data are easily inserted and deleted. It is not necessary to shift elements of a linked list to make room for a new element or to delete an element. On the other hand, because the elements are no longer physically sequenced, we are limited to sequential searches:[1] we cannot use a binary search.[2]

1. Sequential and binary searches are discussed in Chapter 13.
2. When we examine trees, you will see several data structures that allow for easy updates and efficient searches.

Figure 1-3(a) shows a linked list implementation of a linear list. The link in each element, except the last, points to its unique successor; the link in the last element contains a null pointer, indicating the end of the list. Figure 1-3(b) shows a linked list implementation of a non-linear list. An element in a non-linear list can have two or more links. Here each element contains two links, each to one successor. Figure 1-3(c) contains an example of an **empty list**, linear or non-linear. We define an empty list as a null list pointer.

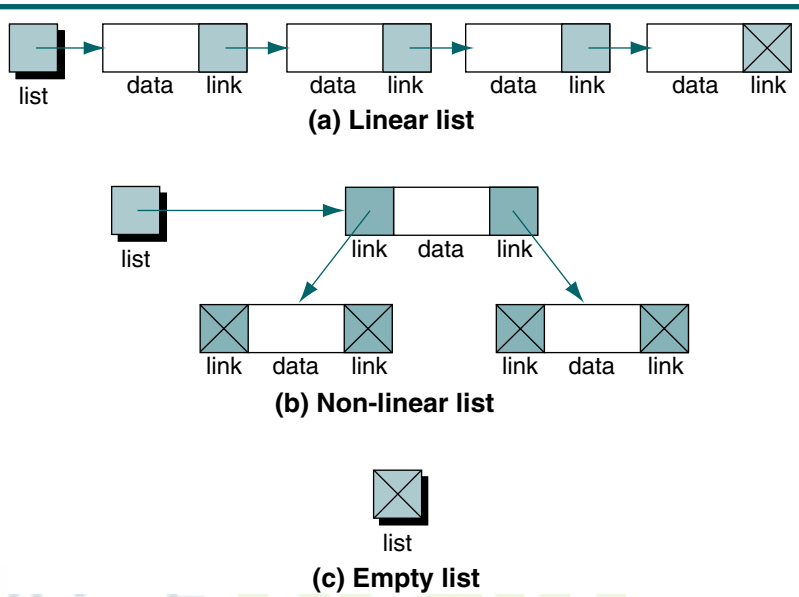


FIGURE 1-3 Linked Lists

In this section, we discuss only the basic concepts for linked lists. We expand on these concepts in future chapters.

Nodes

In linked list implementation, the elements in a list are called nodes. A **node** is a structure that has two parts: the data and one or more links. Figure 1-4 shows two different nodes: one for a linear list and the other for a non-linear list.

The nodes in a linked list are called **self-referential** structures. In a self-referential structure, each instance of the structure contains one or more pointers to other instances of the same structural type. In Figure 1-4, the colored boxes with arrows are the pointers that make the linked list a self-referential structure.

The data part in a node can be a single field, multiple fields, or a structure that contains several fields, but it always acts as a single field. Figure 1-5 shows three designs for a node of a linear list. The upper-left node contains a

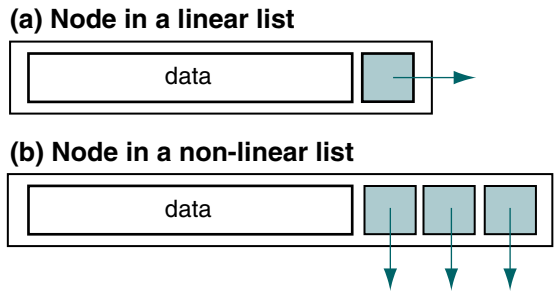


FIGURE 1-4 Nodes

single field, a number, and a link. The upper-right node is more typical. It contains three data fields: a name, an id, and grade points (`grdPts`)—and a link. The third example is the one we recommend. The fields are defined in their own structure, which is then put into the definition of a node structure.

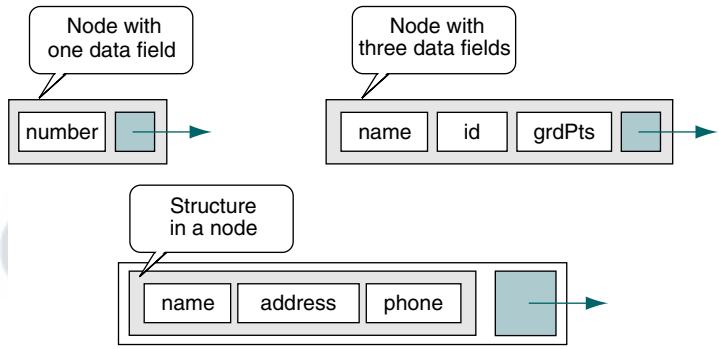


FIGURE 1-5 Linked List Node Structures

Pointers to Linked Lists

A linked list must always have a head pointer. Depending on how we use the list, we may have several other pointers as well. For example, if we are going to search a linked list, we will need an additional pointer to the location where we found the data we were looking for. Furthermore, in many structures, programming is more efficient if there is a pointer to the last node in the list as well as a head pointer.

1.5 Generic Code for ADTs

In data structures we need to create generic code for abstract data types. **Generic code** allows us to write one set of code and apply it to any data type. For

example, we can write generic functions to implement a stack structure. We can then use the generic functions to implement an integer stack, a float stack, a double stack, and so on. Although some high-level languages such as C++ and Java provide special tools to handle generic code, C has limited capability through two features: pointer to *void* and pointer to function.

Pointer to *void*

The first feature is **pointer to *void***. Because C is strongly typed, operations such as assign and compare must use compatible types or be cast to compatible types. The one exception is the pointer to *void*, which can be assigned without a cast. In other words, a pointer to *void* is a generic pointer that can be used to represent any data type during compilation or run time. Figure 1-6 shows the idea of a pointer to *void*. Note that a pointer to *void* is not a null pointer; it is pointing to a generic data type (*void*).

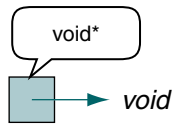


FIGURE 1-6 Pointer to *void*

Example Let us write a simple program to demonstrate the concept. It contains three variables: an integer, a floating-point number, and a *void* pointer. At different times in the program the pointer can be set to the address of the integer value or of the floating-point value. Figure 1-7 shows the situation.

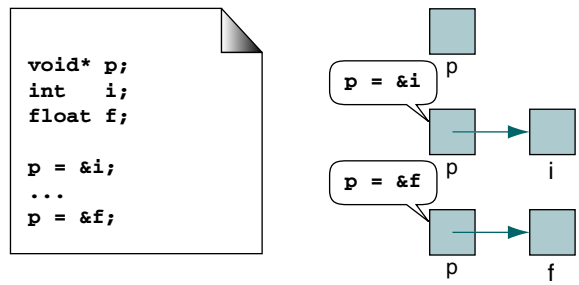


FIGURE 1-7 Pointers for Program 1-1

Program 1-1 uses a pointer to *void* that we can use to print either an integer or a floating-point number.

PROGRAM 1-1 Demonstrate Pointer to *void*

```
 1  /* Demonstrate pointer to void.
 2        Written by:
 3        Date:
 4  */
 5  #include <stdio.h>
 6
 7  int main ()
 8  {
 9  // Local Definitions
10     void* p;
11     int   i = 7 ;
12     float f = 23.5;
13
14  // Statements
15     p = &i;
16     printf ("i contains: %d\n", *((int*)p) );
17
18     p = &f;
19     printf ("f contains: %f\n", *((float*)p));
20
21     return 0;
22  }  // main
```

```
Results:
i contains 7
f contains 23.500000
```

Program 1-1 Analysis

The program is trivial, but it demonstrates the point. The pointer `p` is declared as a *void* pointer, but it can accept the address of an integer or floating-point number. However, we must remember a very import point about pointers to *void*: a pointer to *void* cannot be dereferenced unless it is cast. In other words, we cannot use `*p` without casting. That is why we need to cast the pointer in the print function before we use it for printing.

> A pointer to *void* cannot be dereferenced.

Example

As another example, let us look at a system function, *malloc*. This function returns a pointer to *void*. The designers of the *malloc* function needed to dynamically allocate any type of data. However, instead of using several *malloc* functions, each returning a pointer to a specific data type (`int*`, `float*`, `double*`, and so on), they designed a generic function that returns a pointer to *void* (`void*`). While it is not required, we recommend that the returned pointer be cast to the appropriate type. The following shows the use of *malloc* to create a pointer to an integer.

```
intPtr = (int*)malloc (sizeof (int));
```

Example Now let's look at an example that is similar to what we use to implement our ADTs. We need to have a generic function to create a node structure. The structure has two fields: data and link. The link field is a pointer to the node structure. The data field, however, can be any type: integer, floating point, string, or even another structure. To make the function generic so that we can store any type of data in the node, we use a *void* pointer to data stored in dynamic memory. We declare the node structure as shown in Figure 1-8.

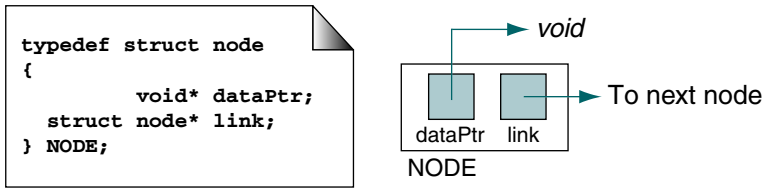


FIGURE 1-8 Pointer to Node

Now let's write the program that calls a function that accepts a pointer to data of any type and creates a node that stores the data pointer and a link pointer. Because we don't know where the link pointer will be pointing, we make it null. The pointer design is shown in Figure 1-9.

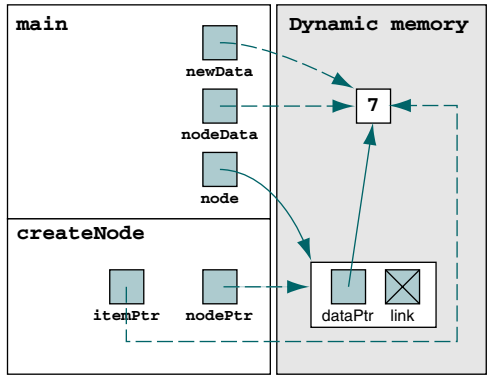


FIGURE 1-9 Pointers for Programs 1-2 and 1-3

Typically, ADTs are stored in their own header files. We begin, therefore, by writing the code for creating the node and placing the code in a header file. This code is shown in Program 1-2.

PROGRAM 1-2 Create Node Header File

```
1  /* Header file for create node structure.
```

continued

PROGRAM 1-2 Create Node Header File *(continued)*

```
2        Written by:
3        Date:
4  */
5  typedef struct node
6  {
7           void* dataPtr;
8    struct node* link;
9  } NODE;
10
11 /* ================== createNode ====================
12    Creates a node in dynamic memory and stores data
13    pointer in it.
14       Pre  itemPtr is pointer to data to be stored.
15       Post node created and its address returned.
16 */
17 NODE* createNode (void* itemPtr)
18 {
19    NODE* nodePtr;
20    nodePtr = (NODE*) malloc (sizeof (NODE));
21    nodePtr->dataPtr = itemPtr;
22    nodePtr->link    = NULL;
23    return nodePtr;
24 }  // createNode
```

Now that we've created the data structure and the create node function, we can write Program 1-3 to demonstrate the use of *void* pointers in a node.

PROGRAM 1-3 Demonstrate Node Creation Function

```
1  /* Demonstrate simple generic node creation function.
2        Written by:
3        Date:
4  */
5  #include <stdio.h>
6  #include <stdlib.h>
7  #include "P1-02.h"                          // Header file
8
9  int main (void)
10 {
11 // Local Definitions
12    int*  newData;
13    int*  nodeData;
14    NODE* node;
15
16 // Statements
17    newData  = (int*)malloc (sizeof (int));
18    *newData = 7;
```

continued

PROGRAM 1-3 Demonstrate Node Creation Function *(continued)*

```
19
20      node = createNode (newData);
21
22      nodeData = (int*)node->dataPtr;
23      printf ("Data from node: %d\n", *nodeData);
24      return 0;
25  }   // main
```

```
Results:
Data from node: 7
```

Program 1-3 Analysis

There are several important concepts in this program. First, the data to be stored in the node is represented by a *void* pointer. Because there are usually many instances of these nodes in a program, the data are stored in dynamic memory. The allocation and storage of the data are the responsibility of the programmer. We show these two steps in statements 17 and 18.

The `createNode` function allocates a node structure in dynamic memory, stores the data *void* pointer in the node, and then returns the node's address. In statement 22, we store the *void* pointer from the node into an integer pointer. Because C is strongly typed, this assignment must be cast to an integer pointer. So, while we can store an address in a *void* pointer without knowing its type, the reverse is not true. To use a *void* pointer, even in an assignment, it must be cast.

> Any reference to a *void* pointer must cast the pointer to the correct type.

Example

ADT structures generally contain several instances of a node. To better demonstrate the ADT concept, therefore, let's modify Program 1-3 to contain two different nodes. In this simple example, we point the first node to the second node. The pointer structure for the program is shown in Figure 1-10.

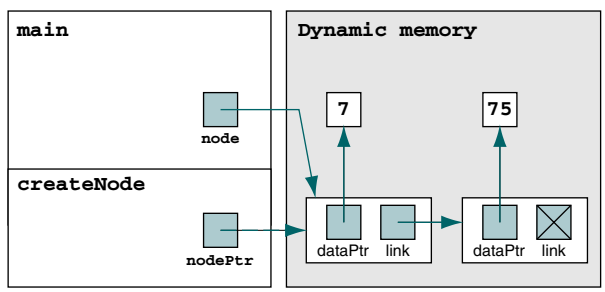


FIGURE 1-10 Structure for Two Linked Nodes

The pointer values in Figure 1-10 represent the settings at the end of Program 1-4.

PROGRAM 1-4 Create List with Two Linked Nodes

```
 1  /* Create a list with two linked nodes.
 2        Written by:
 3        Date:
 4  */
 5  #include <stdio.h>
 6  #include <stdlib.h>
 7  #include "P1-02.h"                        // Header file
 8
 9  int main (void)
10  {
11  // Local Definitions
12     int*  newData;
13     int*  nodeData;
14     NODE* node;
15
16  // Statements
17     newData  = (int*)malloc (sizeof (int));
18     *newData = 7;
19     node = createNode (newData);
20
21     newData    = (int*)malloc (sizeof (int));
22     *newData   = 75;
23     node->link = createNode (newData);
24
25     nodeData = (int*)node->dataPtr;
26     printf ("Data from node 1: %d\n", *nodeData);
27
28     nodeData = (int*)node->link->dataPtr;
29     printf ("Data from node 2: %d\n", *nodeData);
30     return 0;
31  }  // main
```

```
Results:
Data from node 1: 7
Data from node 2: 75
```

Program 1-4 Analysis This program demonstrates an important point. In a generic structure such as shown in the program, the nodes and the data must both be in dynamic memory. When studying the program, follow the code through Figure 1-10.

Pointer to Function

The second tool that is required to create C generic code is pointer to function. In this section we discuss how to use it.

Functions in your program occupy memory. The name of the function is a pointer constant to its first byte of memory. For example, imagine that you have four functions stored in memory: `main`, `fun`, `pun`, and `sun`. This

relationship is shown graphically in Figure 1-11. The name of each function is a pointer to its code in memory.

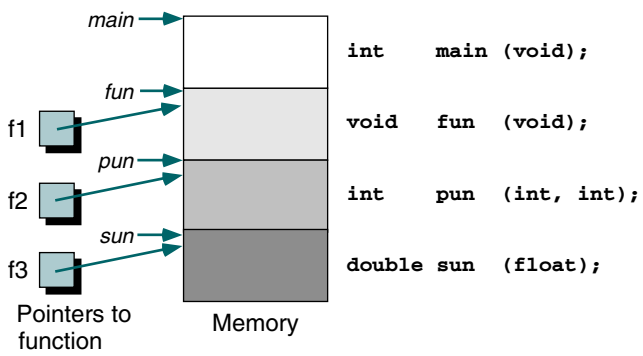


FIGURE 1-11 Functions in Memory

Defining Pointers to Functions

Just as with all other pointer types, we can define pointers to function variables and store the address of `fun`, `pun`, and `sun` in them. To declare a **pointer to function**, we code it as if it were a prototype definition, with the function pointer in parentheses. This format is shown in Figure 1-12. The parentheses are important: without them C interprets the function return type as a pointer.

Using Pointers to Functions

Now that you've seen how to create and use pointers to functions, let's write a generic function that returns the larger of any two pieces of data. The function uses two pointers to *void* as described in the previous section. While our function needs to determine which of the two values represented by the *void* pointers is larger, it cannot directly compare them because it doesn't know what type casts to use with the *void* pointers. Only the application program knows the data types.

The solution is to write simple compare functions for each program that uses our generic function. Then, when we call the generic compare function, we use a pointer to function to pass it the specific compare function that it must use.

Example As we saw in our discussion of pointers to *void*, we place our generic function, which we call `larger`, in a header file so that it can be easily used. The program interfaces and pointers are shown in Figure 1-13.

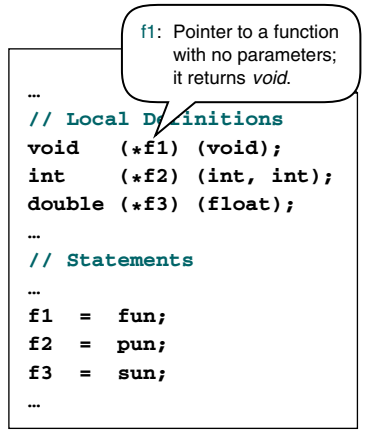


FIGURE 1-12 Pointers to Functions

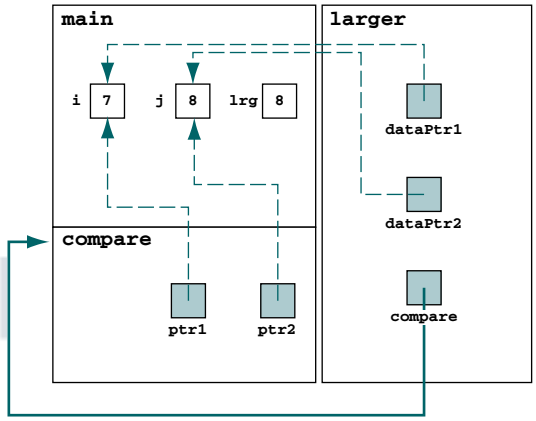


FIGURE 1-13 Design of Larger Function

The code is shown in Program 1-5.

PROGRAM 1-5 Larger Compare Function

```
1  /* Generic function to determine the larger of two
2     values referenced as void pointers.
3        Pre  dataPtr1 and dataPtr2 are pointers to values
4                of an unknown type.
5             ptrToCmpFun is address of a function that
6                knows the data types
7        Post data compared and larger value returned
```

continued

PROGRAM 1-5 Larger Compare Function *(continued)*

```
 8  */
 9  void* larger (void* dataPtr1,    void* dataPtr2,
10                int (*ptrToCmpFun)(void*, void*))
11  {
12     if ((*ptrToCmpFun) (dataPtr1, dataPtr2) > 0)
13           return dataPtr1;
14     else
15           return dataPtr2;
16  }  // larger
```

Program 1-6 contains an example of how to use our generic compare program and pass it a specific compare function.

PROGRAM 1-6 Compare Two Integers

```
 1  /* Demonstrate generic compare functions and pointer to
 2     function.
 3        Written by:
 4        Date:
 5  */
 6  #include <stdio.h>
 7  #include <stdlib.h>
 8  #include "P1-05.h"                        // Header file
 9
10  int   compare (void* ptr1, void* ptr2);
11
12  int main (void)
13  {
14  // Local Definitions
15
16     int i = 7 ;
17     int j = 8 ;
18     int lrg;
19
20  // Statements
21     lrg = (*(int*) larger (&i, &j, compare));
22
23     printf ("Larger value is: %d\n", lrg);
24     return 0;
25  }  // main
26  /* ==================== compare ====================
27     Integer specific compare function.
28        Pre  ptr1 and ptr2 are pointers to integer values
29        Post returns +1 if ptr1 >= ptr2
30             returns -1 if ptr1 <  ptr2
31  */
32  int compare (void* ptr1, void* ptr2)
```

continued

PROGRAM 1-6 Compare Two Integers *(continued)*

```
33  {
34    if (*(int*)ptr1 >=  *(int*)ptr2)
35       return 1;
36    else
37       return -1;
38  }  // compare
```

```
Results:
Larger value is: 8
```

Example Now, let's write a program that compares two floating-point numbers. We can use our larger function, but we need to write a new compare function. We repeat Program 1-6, changing only the compare function and the data-specific statements in *main*. The result is shown in Program 1-7.

PROGRAM 1-7 Compare Two Floating-Point Values

```
 1  /* Demonstrate generic compare functions and pointer to
 2     function.
 3        Written by:
 4        Date:
 5  */
 6  #include <stdio.h>
 7  #include <stdlib.h>
 8  #include "P1-05.h"                    // Header file
 9
10  int   compare (void* ptr1, void* ptr2);
11
12  int main (void)
13  {
14  // Local Definitions
15
16     float f1 = 73.4;
17     float f2 = 81.7;
18     float lrg;
19
20  // Statements
21     lrg = (*(float*) larger (&f1, &f2, compare));
22
23     printf ("Larger value is: %5.1f\n", lrg);
24     return 0;
25  }  // main
26  /* ==================== compare ====================
27     Float specific compare function.
28        Pre  ptr1 and ptr2 are pointers to integer values
29        Post returns +1 if ptr1 >= ptr2
```

continued

PROGRAM 1-7 Compare Two Floating-Point Values *(continued)*

```
30              returns -1 if ptr1 <  ptr2
31  */
32  int compare (void* ptr1, void* ptr2)
33  {
34    if (*(float*)ptr1 >=  *(float*)ptr2)
35       return 1;
36    else
37       return -1;
38  }  // compare
```

```
Results:
Larger value is: 81.7
```

1.6 Algorithm Efficiency

There is seldom a single algorithm for any problem. When comparing two different algorithms that solve the same problem, you often find that one algorithm is an order of magnitude more efficient than the other. In this case, it only makes sense that you be able to recognize and choose the more efficient algorithm.

Although computer scientists have studied algorithms and algorithm efficiency extensively, the field has not been given an official name. Brassard and Bratley coined the term **algorithmics**, which they define as "the systematic study of the fundamental techniques used to design and analyze efficient algorithms."[3] We use the term in this book.

If a function is linear—that is, if it contains no loops or recursions—its efficiency is a function of the number of instructions it contains. In this case, its efficiency depends on the speed of the computer and is generally not a factor in the overall efficiency of a program. On the other hand, functions that use loops or recursion vary widely in efficiency. The study of algorithm efficiency therefore focuses on loops. Our analysis concentrates on loops because recursion can always be converted to a loop.

As we study specific examples, we generally discuss the algorithm's efficiency as a function of the number of elements to be processed. The general format is

$$f(n) = \text{efficiency}$$

The basic concepts are discussed in this section.

3. Gilles Brassard and Paul Bratley, *Algorithmics Theory and Practice* (Englewood Cliffs, N.J.: Prentice Hall, 1988), xiii.

Linear Loops

Let us start with a simple loop. We want to know how many times the body of the loop is repeated in the following code:[4]

```
for (i = 0; i < 1000; i++)
     application code
```

Assuming `i` is an integer, the answer is 1000 times. The number of iterations is directly proportional to the loop factor, 1000. The higher the factor, the higher the number of loops. Because the efficiency is directly proportional to the number of iterations, it is

$$f(n) = n$$

However, the answer is not always as straightforward as it is in the above example. For instance, consider the following loop. How many times is the body repeated in this loop? Here the answer is 500 times. Why?

```
for (i = 0; i < 1000; i += 2)
     application code
```

In this example the number of iterations is half the loop factor. Once again, however, the higher the factor, the higher the number of loops. The efficiency of this loop is proportional to half the factor, which makes it

$$f(n) = n / 2$$

If you were to plot either of these loop examples, you would get a straight line. For that reason they are known as **linear loops**.

Logarithmic Loops

In a linear loop, the loop update either adds or subtracts. In a **logarithmic loop**, the controlling variable is multiplied or divided in each iteration. How many times is the body of the loops repeated in the following program segments?

Multiply Loops	Divide Loops
`for (i = 0; i < 1000; i *= 2)` `  application code`	`for (i = 0; i < 1000; i /= 2)` `  application code`

To help you understand this problem, Table 1-3 analyzes the values of *i* for each iteration. As you can see, the number of iterations is 10 in both cases. The reason is that in each iteration the value of `i` doubles for the multiply loop and is cut in half for the divide loop. Thus, the number of iterations

4. For algorithm efficiency analysis, we use C code so that we can clearly see the looping constructs.

Multiply		Divide	
Iteration	Value of `i`	Iteration	Value of `i`
1	1	1	1000
2	2	2	500
3	4	3	250
4	8	4	125
5	16	5	62
6	32	6	31
7	64	7	15
8	128	8	7
9	256	9	3
10	512	10	1
(exit)	1024	(exit)	0

TABLE 1-3 Analysis of Multiply and Divide Loops

is a function of the multiplier or divisor, in this case 2. That is, the loop continues while the condition shown below is true.

```
multiply   2^Iterations < 1000
divide     1000 / 2^Iterations >= 1
```

Generalizing the analysis, we can say that the iterations in loops that multiply or divide are determined by the following formula:

$$f(n) = \log n$$

Nested Loops

Loops that contain loops are known as **nested loops**. When we analyze nested loops, we must determine how many iterations each loop completes. The total is then the product of the number of iterations in the inner loop and the number of iterations in the outer loop.

```
Iterations = outer loop iterations x inner loop iterations
```

We now look at three nested loops: linear logarithmic, quadratic, and dependent quadratic.

Linear Logarithmic

The inner loop in the following code is a loop that multiplies. To see the multiply loop, look at the update expression in the inner loop.

```
for (i = 0; i < 10; i++)
     for (j = 0; j < 10; j *= 2)
          application code
```

The number of iterations in the inner loop is therefore $\lceil \log 10 \rceil$. However, because the inner loop is controlled by an outer loop, the above formula must be multiplied by the number of times the outer loop executes, which is 10. This gives us

$$10 \log 10$$

which is generalized as

$$f(n) = n \log n$$

Quadratic

In a **quadratic loop**, the number of times the inner loop executes is the same as the outer loop. Consider the following example.

```
for (i = 0; i < 10; i++)
    for (j = 0; j < 10; j++)
         application code
```


The outer loop (*for i*) is executed 10 times. For each of its iterations, the inner loop (*for j*) is also executed 10 times. The answer, therefore, is 100, which is 10×10, the square of the loops. This formula generalizes to

$$f(n) = n^2$$

Dependent Quadratic

In a **dependent quadratic loop**, the number of iterations of the inner loop depends on the outer loop. Consider the nested loop shown in the following example.

```
for (i = 0; i < 10; i++)
    for (j = 0; j < i; j++)
         application code
```

The outer loop is the same as the previous loop. However, the inner loop depends on the outer loop for one of its factors. It is executed only once the first iteration, twice the second iteration, three times the third iteration, and so forth. The number of iterations in the body of the inner loop is calculated as shown below.

```
1 + 2 + 3 + … + 9 + 10 = 55
```

If we compute the average of this loop, it is 5.5 (55/10), which is the same as the number of iterations (10) plus 1 divided by 2. Mathematically, this calculation is generalized to

$$\frac{(n+1)}{2}$$

Multiplying the inner loop by the number of times the outer loop is executed gives us the following formula for a dependent quadratic loop:

$$f(n) = n\left(\frac{n+1}{2}\right)$$

Big-O Notation

With the speed of computers today, we are not concerned with an exact measurement of an algorithm's efficiency as much as we are with its general order of magnitude. If the analysis of two algorithms shows that one executes 15 iterations while the other executes 25 iterations, they are both so fast that we can't see the difference. On the other hand, if one iterates 15 times and the other 1500 times, we should be concerned.

We have shown that the number of statements executed in the function for n elements of data is a function of the number of elements, expressed as $f(n)$. Although the equation derived for a function may be complex, a dominant factor in the equation usually determines the order of magnitude of the result. Therefore, we don't need to determine the complete measure of efficiency, only the factor that determines the magnitude. This factor is the big-O, as in "on the order of," and is expressed as $O(n)$—that is, on the order of n.

This simplification of efficiency is known as big-O analysis. For example, if an algorithm is quadratic, we would say its efficiency is

$$O(n^2)$$

or on the order of n squared.

The **big-O notation** can be derived from $f(n)$ using the following steps:

1. In each term, set the coefficient of the term to 1.

2. Keep the largest term in the function and discard the others. Terms are ranked from lowest to highest as shown below.

$$\log n \quad n \quad n\log n \quad n^2 \quad n^3 \ldots n^k \quad 2^n \quad n!$$

For example, to calculate the big-O notation for

$$f(n) = n\frac{(n+1)}{2} = \frac{1}{2}n^2 + \frac{1}{2}n$$

we first remove all coefficients. This gives us

$$n^2 + n$$

which after removing the smaller factors gives us

$$n^2$$

which in big-O notation is stated as

$$O(f(n)) = O(n^2)$$

To consider another example, let's look at the polynomial expression

$$f(n) = a_j n^k + a_{j-1} n^{k-1} + \ldots + a_2 n^2 + a_1 n + a_0$$

We first eliminate all of the coefficients as shown below.

$$f(n) = n^k + n^{k-1} + \ldots + n^2 + n + 1$$

The largest term in this expression is the first one, so we can say that the order of a polynomial expression is

$$O(f(n)) = O(n^k)$$

Standard Measures of Efficiency

Computer scientists have defined seven categories of algorithm efficiency. We list them in Table 1-4 in order of decreasing efficiency and show the first five of them graphically in Figure 1-14.

Efficiency	Big-O	Iterations	Estimated Time
Logarithmic	$O(\log n)$	14	microseconds
Linear	$O(n)$	10,000	seconds
Linear logarithmic	$O(n(\log n))$	140,000	seconds
Quadratic	$O(n^2)$	$10{,}000^2$	minutes
Polynomial	$O(n^k)$	$10{,}000^k$	hours
Exponential	$O(c^n)$	$2^{10{,}000}$	intractable
Factorial	$O(n!)$	10,000!	intractable

TABLE 1-4 Measures of Efficiency for n = 10,000

Any measure of efficiency presumes that a sufficiently large sample is being considered. If you are dealing with only 10 elements and the time required is a fraction of a second, there is no meaningful difference between two algorithms. On the other hand, as the number of elements being processed grows, the difference between algorithms can be staggering.

Returning for a moment to the question of why we should be concerned about efficiency, consider the situation in which you can solve a problem in three ways: one is linear, another is linear logarithmic, and a third is quadratic. The magnitude of their efficiency for a problem containing 10,000

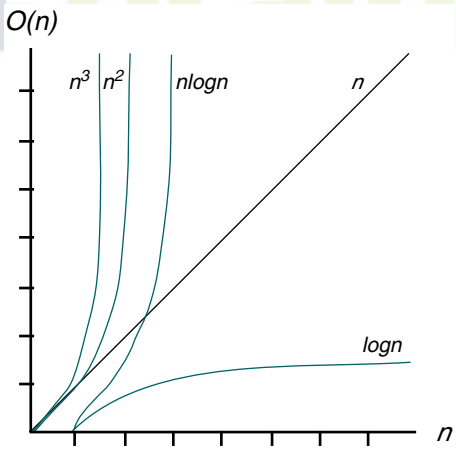


FIGURE 1-14 Plot of Effeciency Measures

elements shows that the linear solution requires a fraction of a second whereas the quadratic solution requires minutes (see Table 1-4).

Looking at the problem from the other end, if we are using a computer that executes a million instructions per second and the loop contains 10 instructions, we spend 0.00001 second for each iteration of the loop. Table 1-4 also contains an estimate of the time needed to solve the problem given different efficiencies.

Big-O Analysis Examples

To demonstrate the concepts we have been discussing, we examine two more algorithms: add and multiply two matrices.

Add Square Matrices

To add two square matrices, we add their corresponding elements; that is, we add the first element of the first matrix to the first element of the second matrix, the second element of the first matrix to the second element of the second matrix, and so forth. Of course, the two matrices must be the same size. This concept is shown in Figure 1-15.

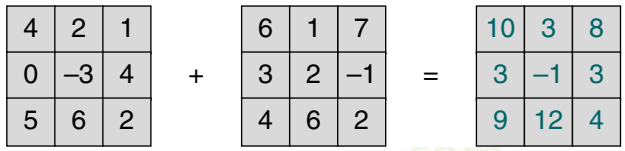


FIGURE 1-15 Add Matrices

The pseudocode to add two matrices is shown in Algorithm 1-3.

ALGORITHM 1-3 Add Two Matrices

```
Algorithm addMatrix (matrix1, matrix2, size, matrix3)
Add matrix1 to matrix2 and place results in matrix3
   Pre  matrix1 and matrix2 have data
        size is number of columns or rows in matrix
   Post matrices added--result in matrix3
1  loop (not end of row)
   1   loop (not end of column)
       1   add matrix1 and matrix2 cells
       2   store sum in matrix3
   2   end loop
2  end loop
end addMatrix
```

Algorithm 1-3 Analysis In this algorithm, we see that for each element in a row, we add all of the elements in a column. This is the classic quadratic loop. The efficiency of the algorithm is therefore $O(\text{size}^2)$ or $O(n^2)$.

Multiply Square Matrices

When two square matrices are multiplied, we must *multiply* each element in a row of the first matrix by its corresponding element in a column of the second matrix. The value in the resulting matrix is then the *sum* of the products. For example, given the matrix in our addition example above, the first element in the resulting matrix—that is, the element at `[0, 0]`—is the sum of the products obtained by multiplying each element in the first *row* (row 0) by its corresponding element in the first *column* (column 0). The value of the element at index location `[0, 1]` is the sum of the products of each element in the first row (again row 0) multiplied by its corresponding element in the second column (column 1). The value of the element at index location `[1, 2]` is the sum of the products of each element in the second *row* multiplied by the corresponding elements in the third *column*. Once again the square matrices must be the same size. Figure 1-16 graphically shows how two matrices are multiplied.

Generalizing this concept, we see that

```
matrix3 [row, col] =
            matrix1[row, 0] x matrix2[0, col]
      +     matrix1[row, 1] x matrix2[1, col]
      +     matrix1[row, 2] x matrix2[2, col]
      ...
      +     matrix1[row, s-1] x matrix2[s-1, col]
where s = size of matrix
```

The pseudocode used for multiplying matrices is provided in Algorithm 1-4.

ALGORITHM 1-4 Multiply Two Matrices

```
Algorithm multiMatrix (matrix1, matrix2, size, matrix3)
Multiply matrix1 by matrix2 and place product in matrix3
   Pre  matrix1 and matrix2 have data
        size is number of columns and rows in matrix
   Post matrices multiplied--result in matrix3
1  loop (not end of row)
   1   loop (not end of column)
       1   loop (size of row times)
           1  calculate sum of
                 (all row cells) * (all column cells)
           2  store sum in matrix3
```

continued

ALGORITHM 1-4 Multiply Two Matrices *(continued)*

```
   2  end loop
2 end loop
3 return
end multiMatrix
```

Algorithm 1-4 Analysis

In this algorithm we see three nested loops. Because each loop starts at the first element, we have a cubic loop. Loops with three nested loops have a big-O efficiency of $O(size^3)$ or $O(n^3)$.

It is also possible to multiply two matrices if the number of rows in the first matrix is the same as the number of columns in the second. We leave the solution to this problem as an exercise (Exercise 21).

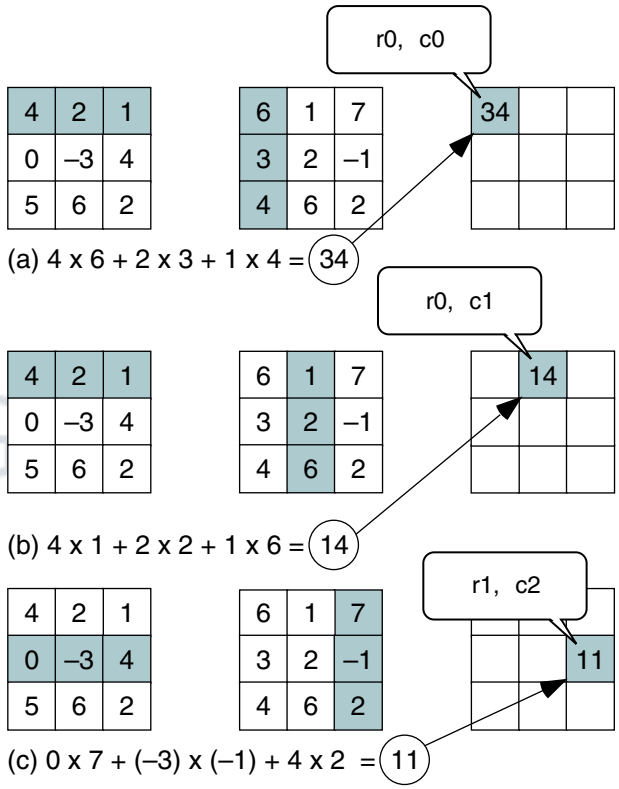


FIGURE 1-16 Multiply Matrices

1.7 Key Terms

abstract data type (ADT)
algorithmics
atomic data
big-O notation
composite data
construct
data
data structure
data type
dependent quadratic loop
empty list
encapsulation
generic code
intelligent data names
linear loop
link
linked list
logarithmic loop
loop
modular programming
nested loop
node
pointer to *void*
pointer to function
pseudocode
quadratic loop
return condition
self-referential
selection statement
sequence
spaghetti code
structured programming

1.8 Summary

- One of the most common tools used to define algorithms is pseudocode.
- Pseudocode is an English-like representation of the code required for an algorithm. It is part English, part structured code.
- Atomic data are data that are single, nondecomposable entities.
- Atomic data types are defined by a set of values and a set of operations that act on the values.
- A data structure is an aggregation of atomic and composite data with a defined relationship.
- An abstract data type (ADT) is a data declaration packaged together with the operations that are meaningful for the data type.
- There are two basic structures used to implement an ADT list: arrays and linked lists.
- In an array, the sequentiality of a list is preserved by the ordered structure of elements. Although searching an array is very efficient, adding and deleting is not.
- Although adding and deleting in a linked list is efficient, searching is not because we must use a sequential search.
- In a linked list, each element contains the location of the next element or elements.

- Abstract data types require generic algorithms, which allow an algorithm to be used with multiple data types.
- The C language has two features that allow the creation of generic code: pointer to *void* and pointer to function.
- A *void* pointer is a generic pointer that can be used to represent any data type.
- A pointer to *void* cannot be dereferenced, which means that nonassignment references to a *void* pointer must be cast to the correct type.
- The name of a function is a pointer constant to the first byte of a function.
- We can use pointer to function as a place holder for the name of a function in a parameter list of a generic function.
- Algorithm efficiency is generally defined as a function of the number of elements being processed and the type of loop being used.
- The efficiency of a logarithmic loop is $f(n) = \log n$.
- The efficiency of a linear loop is $f(n) = n$.
- The efficiency of a linear logarithmic loop is $f(n) = n\ (\log n)$.
- The efficiency of a quadratic loop is $f(n) = n^2$.
- The efficiency of a dependent quadratic loop is $f(n) = n(n + 1)/2$.
- The efficiency of a cubic loop is $f(n) = n^3$.
- The simplification of efficiency is known as big-O notation.
- The seven standard measures of efficiencies are $O(\log n)$, $O(n)$, $O(n(\log n))$, $O(n^2)$, $O(n^k)$, $O(c^n)$, and $O(n!)$.

1.9 Practice Sets

Exercises

1. Structure charts and pseudocode are two different design tools. How do they differ and how are they similar?
2. Using different syntactical constructs, write at least two pseudocode statements to add 1 to a number. For example, any of the following statements could be used to get data from a file:

```
read student file
read student file into student
read (studentFile into student)
```

3. Explain how an algorithm in an application program differs from an algorithm in an abstract data type.
4. Identify the atomic data types for your primary programming language.

5. Identify the composite data types for your primary programming language.
6. Reorder the following efficiencies from smallest to largest:
   a. 2^n
   b. $n!$
   c. n^5
   d. 10,000
   e. $n\log(n)$
7. Reorder the following efficiencies from smallest to largest:
   a. $n\log(n)$
   b. $n + n^2 + n^3$
   c. 24
   d. $n^{0.5}$
8. Determine the big-O notation for the following:
   a. $5n^{5/2} + n^{2/5}$
   b. $6\log(n) + 9n$
   c. $3n^4 + n\log(n)$
   d. $5n^2 + n^{3/2}$
9. Calculate the run-time efficiency of the following program segment:

```
for (i = 1; i <= n; i++)
     printf("%d ", i);
```

10. Calculate the run-time efficiency of the following program segment:

```
for (i = 1; i <= n; i++)
    for (j = 1; j <= n; j++)
        for (k = 1; k <= n; k++)
            print ("%d %d %d\n", i, j, k);
```

11. If the algorithm `doIt` has an efficiency factor of $5n$, calculate the run-time efficiency of the following program segment:

```
for (i = 1, i <= n; i++)
     doIt (...)
```

12. If the efficiency of the algorithm `doIt` can be expressed as $O(n) = n^2$, calculate the efficiency of the following program segment:

```
for (i = 1; i <= n;; i++)
    for (j = 1; j < n, j++)
         doIt (...)
```

13. If the efficiency of the algorithm `doIt` can be expressed as $O(n) = n^2$, calculate the efficiency of the following program segment:

```
for (i = 1; i < n; i *= 2)
    doIt (...)
```

14. Given that the efficiency of an algorithm is $5n^2$, if a step in this algorithm takes 1 nanosecond (10^{-9} seconds), how long does it take the algorithm to process an input of size 1000?

15. Given that the efficiency of an algorithm is n^3, if a step in this algorithm takes 1 nanosecond (10^{-9} seconds), how long does it take the algorithm to process an input of size 1000?
16. Given that the efficiency of an algorithm is $5n\log(n)$, if a step in this algorithm takes 1 nanosecond (10^{-9} seconds), how long does it take the algorithm to process an input of size 1000?
17. An algorithm processes a given input of size n. If n is 4096, the run time is 512 milliseconds. If n is 16,384, the run time is 2048 milliseconds. What is the efficiency? What is the big-O notation?
18. An algorithm processes a given input of size n. If n is 4096, the run time is 512 milliseconds. If n is 16,384, the run time is 8192 milliseconds. What is the efficiency? What is the big-O notation?
19. An algorithm processes a given input of size n. If n is 4096, the run time is 512 milliseconds. If n is 16,384, the run time is 1024 milliseconds. What is the efficiency? What is the big-O notation?
20. Three students wrote algorithms for the same problem. They tested the three algorithms with two sets of data as shown below:
    a. Case 1: $n = 10$
       - Run time for student 1: 1
       - Run time for student 2: 1/100
       - Run time for student 3: 1/1000

    b. Case 2: $n = 100$
       - Run time for student 1: 10
       - Run time for student 2: 1
       - Run time for student 3: 1

    What is the efficiency for each algorithm? Which is the best? Which is the worst? What is the minimum number of test cases (n) in which the best algorithm has the best run time?
21. We can multiply two matrices if the number of columns in the first matrix is the same as the number of rows in the second. Write an algorithm that multiplies an m × n matrix by a n × k matrix.
22. Write a compare function (see Program 1-6) to compare two strings.

Problems

23. Write a pseudocode algorithm for dialing a phone number.
24. Write a pseudocode algorithm for giving all employees in a company a cost-of-living wage increase of 3.2%. Assume that the payroll file includes all current employees.

25. Write a language-specific implementation for the pseudocode algorithm in Problem 24.
26. Write a pseudocode definition for a textbook data structure.
27. Write a pseudocode definition for a student data structure.

Projects

28. Your college bookstore has hired you as a summer intern to design a new textbook inventory system. It is to include the following major processes:

    a. Ordering textbooks
    b. Receiving textbooks
    c. Determining retail price
    d. Pricing used textbooks
    e. Determining quantity on hand
    f. Recording textbook sales
    g. Recording textbook returns

    Write the abstract data type algorithm headers for the inventory system. Each header should include name, parameters, purpose, preconditions, postconditions, and return value types. You may add additional algorithms as required by your analysis.

29. Write the pseudocode for an algorithm that converts a numeric score to a letter grade. The grading scale is the typical absolute scale in which 90% or more is an A, 80% to 89% is a B, 70% to 79% is a C, and 60% to 69% is a D. Anything below 60% is an F.
30. Write the pseudocode for an algorithm that receives an integer and then prints the number of digits in the integer and the sum of the digits. For example, given 12,345 it would print that there are 5 digits with a sum of 15.
31. Write the pseudocode for a program that builds a frequency array for data values in the range 1 to 20 and then prints their histogram. The data are to be read from a file. The design for the program is shown in Figure 1-17.

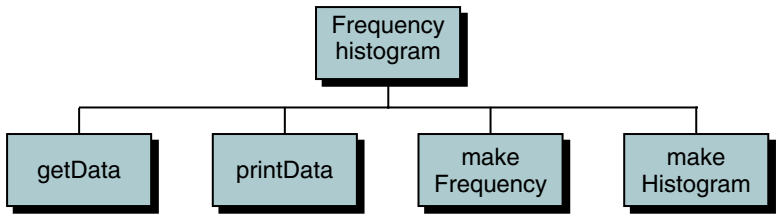


FIGURE 1-17 Design for Frequency Histogram Program

Each of the subalgorithms is described below.

a. The `getData` algorithm reads the file and stores the data in an array.
b. The `printData` algorithm prints the data in the array.
c. The `makeFrequency` algorithm examines the data in the array, one element at a time, and adds 1 to the corresponding element in a frequency array based on the data value.
d. The `makeHistogram` algorithm prints out a vertical histogram using asterisks for each occurrence of an element. For example, if there were five value 1s and eight value 2s in the data, it would print

```
1: *****
2: ********
```

32. Rewrite Program 1-4 to create a list of nodes. Each node consists of two fields. The first field is a pointer to a structure that contains a student id (integer) and a grade-point average (float). The second field is a link. The data are to be read from a text file.

Then write a program to read a file of at least 10 students and test the function you wrote. You will also need to use the generic compare code in Program 1-6 in your program.