Historically, a course on data structures has been a mainstay of most computer science departments. However, over the last 20 years the focus of this course has broadened considerably. The topic of data structures now has been subsumed under the broader topic of abstract data types (ADTs)—the study of classes of objects whose logical behavior is defined by a set of values and a set of operations.

The term abstract data type describes a comprehensive collection of data values and operations; the term data structures refers to the study of data and how to represent data objects within a program—that is, the implementation of structured relationships. The shift in emphasis is representative of the move toward more abstraction in computer science education. We now are interested in the study of the abstract properties of classes of data objects in addition to how the objects might be represented in a program. Johannes J. Martin puts it very succinctly: “. . . depending on the point of view, a data object is characterized by its type (for the user) or by its structure (for the implementor).”

Three Levels of Abstraction

The focus of this book is on abstract data types as viewed from three different perspectives: their specification, their application, and their implementation. The specification describes the logical or abstract level. This level is concerned with what the operations do. The application level, sometimes called the user level, is concerned with how the data type might be used to solve a problem. This level is concerned with why the operations do what they do. The implementation level is where the operations are actually coded. This level is concerned with the how questions.

Within this focus, we stress computer science theory and software engineering principles, including modularization, data encapsulation, information hiding, data

abstraction, object-oriented decomposition, functional decomposition, the analysis of algorithms, and life-cycle software verification methods. We feel strongly that these principles should be introduced to computer science students early in their education so that they learn to practice good software techniques from the beginning.

An understanding of theoretical concepts helps students put the new ideas they encounter into place, and practical advice allows them to apply what they have learned. To teach these concepts to students who may not have completed many college-level mathematics courses, we consistently use intuitive explanations, even for topics that have a basis in mathematics, like the analysis of algorithms. In all cases, our highest goal has been to make our explanations as readable and as easily understandable as possible.

Prerequisite Assumptions

In this book, we assume that the students are familiar with the following C++ constructs.

- Built-in simple data types
- Stream I/O as provided in <iostream>
- Stream I/O as provided in <fstream>
- Control structures while, do-while, for, if, and switch
- User-defined functions with value and reference parameters
- Built-in array types
- Class construct

We have included sidebars within the text to refresh the students’ memories concerning some of the details of these topics.

Changes in the Fourth Edition

The fourth edition incorporates the following changes:

*Array-based and linked implementations for each ADT are presented together.* In previous editions, each ADT was presented using an array-based implementation. Then the linked implementations were gathered together in a chapter. You, our readers, suggested that both implementations of an ADT should be shown together. We have taken your advice.

*Unsorted and sorted lists have been separated into two chapters.* Chapter 3 covers both implementations of the ADT Unsorted List. Chapter 4 covers both implementations of the ADT Sorted List. Dynamically allocated arrays are now covered in Chapter 4, between the array-based and linked-based implementations.

*Templates have been moved to Chapter 6.* Generics are an important feature in programming; however, the template construct can be confusing to students. Chapters 3 through 5 provide the generic abstraction by having the programs include files that define the
items on the structure. Templates are discussed in Chapter 6, after the presentation of lists, stacks, and queues. Overloading operators has also been moved to Chapter 6, immediately following the discussion of copy constructors.

Complexity analysis is moved into Chapter 2. Complexity analysis, using Big-O notation, has been moved to the end of Chapter 2, following the discussion of the abstraction inherent in high-level languages. This placement emphasizes that complexity analysis is a tool of software engineering.

New section added on bounded and unbounded structures. Because we talk about array-based lists as being bound and linked lists as being unbound, we give the students the impression that bound versus unbound is strictly an implementation issue. We have added a section that defines bound and unbound in terms of logical issues as well as implementation issues.

New case studies are provided in Chapters 2 and 7. A complete Date ADT is written as a case study in Chapter 2. This ADT includes a function to calculate a date a certain number of days away and a function to print the month as a string. These methods use an auxiliary array to hold information, thus showing the trade-off between storing data and calculating it. Quicksort, the case study in Chapter 7, has been moved to Chapter 10. The new case study in Chapter 7 involves escaping from a maze.

Content and Organization

Chapter 1 outlines the basic goals of high-quality software and the basic principles of software engineering for designing and implementing programs to meet these goals. Abstraction, functional decomposition, and object-oriented design are discussed. This chapter also addresses what we see as a critical need in software education: the ability to design and implement correct programs and to verify that they are actually correct. Topics covered include the concept of “life-cycle” verification; designing for correctness using preconditions and postconditions; the use of deskchecking and design/code walkthroughs and inspections to identify errors before testing; debugging techniques, data coverage (black box), and code coverage (clear or white box) approaches; and test plans, unit testing, and structured integration testing using stubs and drivers. The concept of a generalized test driver is presented and executed in a case study that develops the ADT Fraction.

Chapter 2 presents data abstraction and encapsulation, the software engineering concepts that relate to the design of the data structures used in programs. Three perspectives of data are discussed: abstraction, implementation, and application. These perspectives are illustrated using a real-world example (a library) and then are applied to built-in data structures that C++ supports: structs and arrays. The C++ class type is presented as the way to represent the abstract data types we examine in subsequent chapters. The principles of object-oriented programming—encapsulation, inheritance, and polymorphism—are introduced here along with the accompanying C++ implementation constructs. The case study at the end of this chapter reinforces the ideas of data abstraction and encapsulation in designing and implementing a user-defined data type representing a date. This class is tested using a version of the generalized test driver.
Chapter 2 includes a discussion of two C++ constructs that help users write better software: namespace and exception handling using the try/catch statement. Various approaches to error handling are demonstrated in subsequent chapters.

Because there is more than one way to solve a problem, we discuss how competing solutions can be compared through the analysis of algorithms, using Big-O notation. Throughout the rest of the book, the ADT implementations are compared using Big-O notation.

We would like to think that the material in Chapters 1 and 2 is a review for most students. However, the concepts in these two chapters are so crucial to the future of any and all students that we feel we cannot rely on their having seen the material before.

Chapter 3 introduces the most fundamental abstract data type of them all: the unsorted list. The chapter begins with a general discussion of operations on abstract data types and then presents the framework with which all of the other data types are examined: a presentation and discussion of the specification, a brief application using the operations, and the design and coding of the operations. The specification is of the ADT Unsorted List. An array-based implementation of the specification is developed.

The concept of dynamic allocation is introduced, along with the syntax for using C++ pointer variables. The concept of linking nodes together to form lists is presented in clear detail with many diagrams. This technique is then used to re-implement the unsorted list. The array-based and linked implementations are compared using Big-O notation.

Chapter 4 introduces the ADT Sorted List and develops the array-based implementation. The binary search is introduced as a way to improve the performance of the search operation in the sorted list. The ADT is then implemented using a linked implementation. These implementations are compared using Big-O notation.

Both the logical and physical distinctions between bound and unbound structures are discussed. The four-phase object-oriented methodology is presented and demonstrated in the case study that takes a simple real estate database.

Chapter 5 introduces the ADTs Stack and Queue. Each ADT is first considered from its abstract perspective, and the idea of recording the logical abstraction in an ADT specification is stressed. The operations are used in an application program; then the set of operations is implemented in C++ using an array-based implementation, followed by a linked implementation. A case study using stacks (postfix expression evaluator) and one using queues (simulation) are presented.

Chapter 6 is a collection of advanced concepts and techniques. Templates are introduced as a way of implementing generic classes. Circular linked lists and doubly linked lists are discussed. The insertion, deletion, and list traversal algorithms are developed and implemented for each variation. An alternative representation of a linked structure, using static allocation (an array of structs), is designed. Class copy constructors, operator overloading, and dynamic binding are covered in detail. The case study uses doubly linked lists to implement large integers.

Chapter 7 presents recursion, giving the student an intuitive understanding of the concept, and then shows how recursion can be used to solve programming problems.
Guidelines for writing recursive functions are illustrated with many examples. After demonstrating that a by-hand simulation of a recursive routine can be very tedious, a simple three-question technique is introduced for verifying the correctness of recursive functions. Because many students are wary of recursion, the introduction to this material is deliberately intuitive and nonmathematical. A more detailed discussion of how recursion works leads to an understanding of how recursion can be replaced with iteration and stacks. The case study develops and implements the process of escaping from a maze.

**Chapter 8** introduces binary search trees as a way to arrange data, giving the flexibility of a linked structure with $O(\log N)$ insertion and deletion time. In order to build on the previous chapter and exploit the inherent recursive nature of binary trees, the algorithms first are presented recursively. After all the operations have been implemented recursively, we code the insertion and deletion operations iteratively to show the flexibility of binary search trees. A nonlinked array-based binary tree implementation is described. The case study discusses the process of building an index for a manuscript and implements the first phase.

**Chapter 9** presents a collection of other branching structures: priority queues (implemented with both lists and heaps), graphs, and sets. The graph algorithms make use of stacks, queues, and priority queues, thus both reinforcing earlier material and demonstrating how general these structures are. Two set implementations are discussed: the bit-vector representation, in which each item in the base set is assigned a present/absent flag and the operations are the built-in logic operations, and a list-based representation, in which each item in a set is represented in a list of set items. If the item is not in the list, it is not in the set.

**Chapter 10** presents a number of sorting and searching algorithms and asks the question: Which are better? The sorting algorithms that are illustrated, implemented, and compared include straight selection sort, two versions of bubble sort, Quicksort, heap sort, and merge sort. The sorting algorithms are compared using Big-O notation. The discussion of algorithm analysis continues in the context of searching. Previously presented searching algorithms are reviewed and new ones are described. Hashing techniques are discussed in some detail. Finally, radix sort is presented and analyzed.

**Additional Features**

*Chapter Goals* A set of goals presented at the beginning of each chapter helps the students assess what they have learned. These goals are tested in the exercises at the end of each chapter.

*Chapter Exercises* Most chapters have more than 35 exercises. They vary in levels of difficulty, including short programming problems, the analysis of algorithms, and problems to test the students’ understanding of concepts. Approximately one-third of the exercises are answered in the back of the book. The answer key for the remaining exercises is in the *Instructor’s Guide.*
Case Studies  There are seven case studies. Each includes a problem description, an analysis of the problem input and required output, and a discussion of the appropriate data types to use. Several of the case studies are completely coded and tested. Others are left at various stages in their development, requiring the student to complete and test the final version.

Program Disk  The specification and implementation of each class representing an ADT is available on a program disk that can be downloaded, free of charge, from the Jones and Bartlett student page on the World Wide Web (www.jbpub.com/). The source code for the completed case studies and the partial source code for the others is also available.

Programming Assignments  A collection of recommended programming assignments for each chapter is included at the end of the book. The assignments represent a range of difficulty levels and were carefully chosen to illustrate the techniques described in the text. A large selection of additional programming assignments is also available in the Instructor’s Guide.

Instructor Support Material  An Instructor’s Guide is available online and includes the following sections for each chapter:

- Goals
- Outline
- Teaching Notes: suggestions for how to teach the material covered in the chapter
- Workouts: suggestions for in-class activities, discussion questions, and short exercises
- Quickie Quiz Questions: additional short-answer questions that can be used in class to test student comprehension
- Exercise Key: answers to those questions that are not in the back of the book
- Programs: suitable programs in ready-to-copy format
- An ITK, which includes
  - Electronic test bank
  - PowerPoint Presentations
  - Electronic Instructor’s Guide

The URL is: http://computerscience.jbpub.com/cppdatastructures/instructor_resources.cfm.

Acknowledgments

First we would like to thank the twenty-six people who replied to our Web survey concerning this new edition. Respondents included both users and non-users of one of the previous editions. Your comments were invaluable: Thank you. We went back to some of the respondents for more information. Six of you graciously answered an additional survey. Again, thank you.
Finally, we very much appreciate the reviewers for this new edition.

Isabeth Cominsky, Mohawk Valley Community College
Francine Federman, Farmingdale State University of New York
Bruce Maxim, University of Michigan-Dearborn

A special thanks goes to John McCormick, University of Northern Iowa, and Dan Joyce at Villanova. John and Dan graciously allowed us to use some of their analogies and descriptions from *Ada Plus Data Structures* and *Object-Oriented Data Structures Using Java*, respectively. Thanks also to the students at Uppsala University in Sweden who used the final draft of the second edition manuscript in a course in the fall of 1997. Because non-English readers see what is written, not what they expect to see, their comments were invaluable in cleaning up ambiguous wording.

Thanks to my husband Al, our children and grandchildren too numerous to name, and our dogs Winston, who barks at the FedEx delivery person, and Chrissie, whose role in life is to keep the house in turmoil, mud, and tennis balls. Thanks to my tennis friends, who kept me fit, and my bridge friends, who challenged my mind.

A virtual bouquet of roses to the people who have worked on this book: David and Sigrid Wile, along with our Jones and Bartlett family: Tim Anderson, my editor, and Amy Rose, my production goddess.

N. D.