Preface

We study data structures so that we can learn to write more efficient programs. But why must programs be efficient when new computers are faster every year? The reason is that our ambitions grow with our capabilities. Instead of rendering efficiency needs obsolete, the modern revolution in computing power and storage capability merely raises the efficiency stakes as we computerize more complex tasks.

The quest for program efficiency need not and should not conflict with sound design and clear coding. Creating efficient programs has little to do with "programming tricks" but rather is based on good organization of information and good algorithms. A programmer who has not mastered the basic principles of clear design is not likely to write efficient programs. Conversely, "software engineering" cannot be used as an excuse to justify inefficient performance. Generality in design can and should be achieved without sacrificing performance, but this can only be done if the designer understands how to measure performance and does so as an integral part of the design and implementation process. Most computer science curricula recognize that good programming skills begin with a strong emphasis on fundamental software engineering principles. Then, once a programmer has learned the principles of clear program design and implementation, the next step is to study the effects of data organization and algorithms on program efficiency.

Approach: This book describes many techniques for representing data. These techniques are presented within the context of the following principles:

1. Each data structure and each algorithm has costs and benefits. Practitioners need a thorough understanding of how to assess costs and benefits to be able to adapt to new design challenges. This requires an understanding of the principles of algorithm analysis, and also an appreciation for the significant effects of the physical medium employed (e.g., data stored on disk versus main memory).

2. Related to costs and benefits is the notion of tradeoffs. For example, it is quite common to reduce time requirements at the expense of an increase in space requirements, or vice versa. Programmers face tradeoff issues regularly in all phases of software design and implementation, so the concept must become deeply ingrained.
3. Programmers should know enough about common practice to avoid reinventing the wheel. Thus, programmers need to learn the commonly used data structures, their related algorithms, and the most frequently encountered design patterns found in programming.
4. Data structures follow needs. Programmers must learn to assess application needs first, then find a data structure with matching capabilities. To do this requires competence in principles 1, 2, and 3.

As I have taught data structures through the years, I have found that design issues have played an ever greater role in my courses. This can be traced through the various editions of this textbook by the increasing coverage for design patterns and generic interfaces. The first edition had no mention of design patterns. The second edition had limited coverage of a few example patterns, and introduced the dictionary ADT and comparator classes. With the third edition, there is explicit coverage of some design patterns that are encountered when programming the basic data structures and algorithms covered in the book.

Using the Book in Class: Data structures and algorithms textbooks tend to fall into one of two categories: teaching texts or encyclopedias. Books that attempt to do both usually fail at both. This book is intended as a teaching text. I believe it is more important for a practitioner to understand the principles required to select or design the data structure that will best solve some problem than it is to memorize a lot of textbook implementations. Hence, I have designed this as a teaching text that covers most standard data structures, but not all. A few data structures that are not widely adopted are included to illustrate important principles. Some relatively new data structures that should become widely used in the future are included.

Within an undergraduate program, this textbook is designed for use in either an advanced lower division (sophomore or junior level) data structures course, or for a senior level algorithms course. New material has been added in the third edition to support its use in an algorithms course. Normally, this text would be used in a course beyond the standard freshman level "CS2" course that often serves as the initial introduction to data structures. Readers of this book should have programming experience, typically two semesters or the equivalent of a structured programming language such as Pascal or **C**, and including at least some exposure to Java. Readers who are already familiar with recursion will have an advantage. Students of

data structures will also benefit from having first completed a good course in Discrete Mathematics. Nonetheless, Chapter 2 attempts to give a reasonably complete survey of the prerequisite mathematical topics at the level necessary to understand their use in this book. Readers may wish to refer back to the appropriate sections as needed when encountering unfamiliar mathematical material.

A sophomore-level class where students have only a little background in basic data structures or analysis (that is, background equivalent to what would be had from a traditional CS2 course) might cover Chapters 1-11 in detail, as well as selected topics from Chapter 13. That is how I use the book for my own sophomore-level class. Students with greater background might cover Chapter 1, skip most of Chapter 2 except for reference, briefly cover Chapters 3 and 4, and then cover chapters 5-12 in detail. Again, only certain topics from Chapter 13 might be covered, depending on the programming assignments selected by the instructor. A senior-level algorithms course would focus on Chapters 11 and 14-17.

Chapter 13 is intended in part as a source for larger programming exercises. I recommend that all students taking a data structures course be required to implement some advanced tree structure, or another dynamic structure of comparable difficulty such as the skip list or sparse matrix representations of Chapter 12. None of these data structures are significantly more difficult to implement than the binary search tree, and any of them should be within a student's ability after completing Chapter 5.

While I have attempted to arrange the presentation in an order that makes sense, instructors should feel free to rearrange the topics as they see fit. The book has been written so that once the reader has mastered Chapters 1-6, the remaining material has relatively few dependencies. Clearly, external sorting depends on understanding internal sorting and disk files. Section 6.2 on the UNION/FIND algorithm is used in Kruskal's Minimum-Cost Spanning Tree algorithm. Section 9.2 on self-organizing lists mentions the buffer replacement schemes covered in Section 8.3. Chapter 14 draws on examples from throughout the book. Section 17.2 relies on knowledge of graphs. Otherwise, most topics depend only on material presented earlier within the same chapter.

Most chapters end with a section entitled "Further Reading." These sections are not comprehensive lists of references on the topics presented. Rather, I include books and articles that, in my opinion, may prove exceptionally informative or entertaining to the reader. In some cases I include references to works that should become familiar to any well-rounded computer scientist.

Use of Java: The programming examples are written in Java™. As with any programming language, Java has both advantages and disadvantages. Java is a

small language. There usually is only one way to do something, and this has the happy tendency of encouraging a programmer toward clarity when used correctly. In this respect, it is superior to **C** or **C**++. Java serves nicely for defining and using most traditional data structures such as lists and trees. On the other hand, Java is quite poor when used to do file processing, being both cumbersome and inefficient. It is also a poor language when fine control of memory is required. As an example, applications requiring memory management, such as those discussed in Section 12.3, are difficult to write in Java. Since I wish to stick to a single language throughout the text, like any programmer I must take the bad along with the good. The most important issue is to get the ideas across, whether or not those ideas are natural to a particular language of discourse. Most programmers will use a variety of programming languages throughout their career, and the concepts described in this book should prove useful in a variety of circumstances.

I do not wish to discourage those unfamiliar with Java from reading this book. I have attempted to make the examples as clear as possible while maintaining the advantages of Java. Java is used here strictly as a tool to illustrate data structures concepts. Fortunately, Java is an easy language for **C** or Pascal programmers to read with a minimal amount of study of the syntax related to object-oriented programming. In particular, I make use of Java's support for hiding implementation details, including features such as classes, private class members, and interfaces. These features of the language support the crucial concept of separating logical design, as embodied in the abstract data type, from physical implementation as embodied in the data structure.

I make no attempt to teach Java within the text. An Appendix is provided that describes the Java syntax and concepts necessary to understand the program examples. I also provide the actual Java code used in the text through anonymous FTP.

Inheritance, a key feature of object-oriented programming, is used only sparingly in the code examples. Inheritance is an important tool that helps programmers avoid duplication, and thus minimize bugs. From a pedagogical standpoint, however, inheritance often makes code examples harder to understand since it tends to spread the description for one logical unit among several classes. Thus, some of my class definitions for objects such as tree or list nodes do not take full advantage of possible inheritance from earlier code examples. This does not mean that a programmer should do likewise. Avoiding code duplication and minimizing errors are important goals. Treat the programming examples as illustrations of data structure principles, but do not copy them directly into your own programs.

My Java implementations serve to provide concrete illustrations of data structure principles. They are not meant to be a series of commercial-quality Java class implementations. The code examples provide less parameter checking than is sound programming practice for commercial programmers. Some parameter checking is included in the form of calls to methods in class **`Assert`**. These methods are modeled after the **C** standard library function **`assert`**. Method **`Assert.notFalse`** takes a Boolean expression. If this expression evaluates to **`false`**, then the program terminates immediately. Method **`Assert.notNull`** takes a reference to class **`Object`**, and terminates the program if the value of the reference is **`null`**. (To be precise, these functions throw an **`IllegalArgumentException`**, which typically results in terminating the program unless the programmer takes action to handle the exception.) Terminating a program when a function receives a bad parameter is generally considered undesirable in real programs, but is quite adequate for understanding how a data structure is meant to operate. In real programming applications, Java's exception handling features should be used to deal with input data errors.

I make a distinction in the text between "Java implementations" and "pseudocode." Code labeled as a Java implementation has actually been compiled and tested on one or more Java compilers. Pseudocode examples often conform closely to Java syntax, but typically contain one or more lines of higher level description. Pseudocode is used where I perceived a greater pedagogical advantage to a simpler, but less precise, description.

Exercises and Projects: Proper implementation and anaysis of data structures cannot be learned simply by reading a book. You must practice by implementing real programs, constantly comparing different techniques to see what really works best in a given situation.

One of the most important aspects of a course in data structures is that it is where students really learn to program using pointers and dynamic memory allocation, by implementing data structures such as linked lists and trees. Its also where students truly learn recursion. In our curriculum, this is the first course where students do significant design, because it often requires real data structures to motivate significant design exercises. Finally, the fundamental differences between memory-based and disk-based data access cannot be appreciated without practical programming experience. For all of these reasons, a data structures course cannot succeed without a significant programming component. In our department, the data structures course is arguably the most difficult programming course in the curriculum.

Students should also work problems to develop their analytical abilities. I provide over 400 exercises and suggestions for programming projects. I urge readers to take advantage of them.

Contacting the Author and Supplementary Materials: A book such as this is sure to contain errors and have room for improvement. I welcome bug reports and constructive criticism. I can be reached by electronic mail via the Internet at **shaffer@vt.edu**. Alternatively, comments can be mailed to

Cliff Shaffer
Department of Computer Science
Virginia Tech
Blacksburg, VA 24061

A set of transparency masters for use in conjunction with this book can be obtained via the WWW at `http://www.cs.vt.edu/~shaffer/book.html`. The Java code examples are also available this site. Online Web pages for Virginia Tech's sophomore-level data structures class can be found at URL

http://courses.cs.vt.edu/~cs3114

This book was originally typeset by the author with LaTeX. The bibliography was prepared using BibTeX. The index was prepared using **makeindex**. The figures were mostly drawn with **Xfig**. Figures 3.1 and 9.8 were partially created using Mathematica.

Acknowledgments: It takes a lot of help from a lot of people to make a book. I wish to acknowledge a few of those who helped to make this book possible. I apologize for the inevitable omissions.

Virginia Tech helped make this whole thing possible through sabbatical research leave during Fall 1994, enabling me to get the project off the ground. My department heads during the time I have written the various editions of this book, Dennis Kafura and Jack Carroll, provided unwavering moral support for this project. Mike Keenan, Lenny Heath, and Jeff Shaffer provided valuable input on early versions of the chapters. I also wish to thank Lenny Heath for many years of stimulating discussions about algorithms and analysis (and how to teach both to students). Steve Edwards deserves special thanks for spending so much time helping me on various redesigns of the C++ and Java code versions for the second and third editions, and many hours of discussion on the principles of program design. Thanks to Layne Watson for his help with Mathematica, and to Bo Begole, Philip Isenhour, Jeff Nielsen, and Craig Struble for much technical assistance. Thanks to Bill McQuain, Mark Abrams and Dennis Kafura for answering lots of silly questions about C++ and Java.

I am truly indebted to the many reviewers of the various editions of this manuscript. For the first edition these reviewers included J. David Bezek (University of Evansville), Douglas Campbell (Brigham Young University), Karen Davis (University of Cincinnati), Vijay Kumar Garg (University of Texas – Austin), Jim Miller (University of Kansas), Bruce Maxim (University of Michigan – Dearborn), Jeff Parker (Agile Networks/Harvard), Dana Richards (George Mason University), Jack Tan (University of Houston), and Lixin Tao (Concordia University). Without their help, this book would contain many more technical errors and many fewer insights.

For the second edition, I wish to thank these reviewers: Gurdip Singh (Kansas State University), Peter Allen (Columbia University), Robin Hill (University of Wyoming), Norman Jacobson (University of California – Irvine), Ben Keller (Eastern Michigan University), and Ken Bosworth (Idaho State University). In addition, I wish to thank Neil Stewart and Frank J. Thesen for their comments and ideas for improvement.

Third edition reviewers included Randall Lechlitner (University of Houstin, Clear Lake) and Brian C. Hipp (York Technical College). I thank them for their comments.

Without the hard work of many people at Prentice Hall, none of this would be possible. Authors simply do not create printer-ready books on their own. Foremost thanks go to Kate Hargett, Petra Rector, Laura Steele, and Alan Apt, my editors over the years. My production editors, Irwin Zucker for the second edition, Kathleen Caren for the original C++ version, and Ed DeFelippis for the Java version, kept everything moving smoothly during that horrible rush at the end. Thanks to Bill Zobrist and Bruce Gregory (I think) for getting me into this in the first place. Others at Prentice Hall who helped me along the way include Truly Donovan, Linda Behrens, and Phyllis Bregman. I am sure I owe thanks to many others at Prentice Hall for their help in ways that I am not even aware of.

I wish to express my appreciation to Hanan Samet for teaching me about data structures. I learned much of the philosophy presented here from him as well, though he is not responsible for any problems with the result. Thanks to my wife Terry, for her love and support, and to my daughters Irena and Kate for pleasant diversions from working too hard. Finally, and most importantly, to all of the data structures students over the years who have taught me what is important and what should be skipped in a data structures course, and the many new insights they have provided. This book is dedicated to them.

Clifford A. Shaffer
Blacksburg, Virginia