

---

# Data Structures and Algorithms with Object-Oriented Design Patterns in Ruby

---



---

# Data Structures and Algorithms with Object-Oriented Design Patterns in Ruby

---

Bruno R. Preiss  
B.A.Sc., M.A.Sc., Ph.D., P.Eng.

MMIV

Copyright © 2004 by Bruno R. Preiss.

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior written permission of the author.

This book was prepared with L<sup>A</sup>T<sub>E</sub>X and reproduced from camera-ready copy supplied by the author. The book is typeset using the Computer Modern fonts designed by Donald E. Knuth with various additional glyphs designed by the author and implemented using METAFONT.

METAFONT is a trademark of Addison Wesley Publishing Company.

T<sub>E</sub>X is a trademark of the American Mathematical Society.

UNIX is a registered trademark of AT&T Bell Laboratories.

Microsoft is a registered trademark of Microsoft Corporation.

*To Patty*



# Contents

<b>Preface</b>	<b>xi</b>
<b>1 Introduction</b>	<b>1</b>
1.1 What This Book Is About . . . . .	1
1.2 Object-Oriented Design . . . . .	1
1.3 Object Hierarchies and Design Patterns . . . . .	2
1.4 The Features of Ruby You Need to Know . . . . .	3
1.5 How This Book Is Organized . . . . .	5
<b>2 Algorithm Analysis</b>	<b>7</b>
2.1 A Detailed Model of the Computer . . . . .	8
2.2 A Simplified Model of the Computer . . . . .	21
Exercises . . . . .	30
Programming Projects . . . . .	32
<b>3 Asymptotic Notation</b>	<b>35</b>
3.1 An Asymptotic Upper Bound—Big Oh . . . . .	35
3.2 An Asymptotic Lower Bound—Omega . . . . .	45
3.3 More Notation—Theta and Little Oh . . . . .	48
3.4 Asymptotic Analysis of Algorithms . . . . .	48
Exercises . . . . .	60
Programming Projects . . . . .	63
<b>4 Foundational Data Structures</b>	<b>65</b>
4.1 Ruby Arrays . . . . .	65
4.2 Multi-Dimensional Arrays . . . . .	70
4.3 Singly-Linked Lists . . . . .	77
Exercises . . . . .	85
Programming Projects . . . . .	86
<b>5 Data Types and Abstraction</b>	<b>89</b>
5.1 Abstract Data Types . . . . .	89
5.2 Design Patterns . . . . .	90
Exercises . . . . .	104
Programming Projects . . . . .	106

---

<b>6</b>	<b>Stacks, Queues, and Deques</b>	<b>107</b>
6.1	Stacks . . . . .	107
6.2	Queues . . . . .	119
6.3	Deques . . . . .	126
	Exercises . . . . .	134
	Programming Projects . . . . .	136
<b>7</b>	<b>Ordered Lists and Sorted Lists</b>	<b>139</b>
7.1	Ordered Lists . . . . .	139
7.2	Sorted Lists . . . . .	161
	Exercises . . . . .	171
	Programming Projects . . . . .	173
<b>8</b>	<b>Hashing, Hash Tables, and Scatter Tables</b>	<b>175</b>
8.1	Hashing—The Basic Idea . . . . .	175
8.2	Hashing Methods . . . . .	178
8.3	Hash Function Implementations . . . . .	182
8.4	Hash Tables . . . . .	189
8.5	Scatter Tables . . . . .	195
8.6	Scatter Table using Open Addressing . . . . .	204
8.7	Applications . . . . .	215
	Exercises . . . . .	217
	Programming Projects . . . . .	219
<b>9</b>	<b>Trees</b>	<b>221</b>
9.1	Basics . . . . .	222
9.2	$N$ -ary Trees . . . . .	225
9.3	Binary Trees . . . . .	228
9.4	Tree Traversals . . . . .	229
9.5	Expression Trees . . . . .	231
9.6	Implementing Trees . . . . .	233
	Exercises . . . . .	257
	Programming Projects . . . . .	259
<b>10</b>	<b>Search Trees</b>	<b>261</b>
10.1	Basics . . . . .	261
10.2	Searching a Search Tree . . . . .	263
10.3	Average Case Analysis . . . . .	264
10.4	Implementing Search Trees . . . . .	270
10.5	AVL Search Trees . . . . .	275
10.6	$M$ -Way Search Trees . . . . .	287
10.7	B-Trees . . . . .	294
10.8	Applications . . . . .	304
	Exercises . . . . .	305
	Programming Projects . . . . .	308

---

<b>11 Heaps and Priority Queues</b>	<b>309</b>
11.1 Basics . . . . .	309
11.2 Binary Heaps . . . . .	312
11.3 Leftist Heaps . . . . .	321
11.4 Binomial Queues . . . . .	327
11.5 Applications . . . . .	341
Exercises . . . . .	345
Programming Projects . . . . .	347
<b>12 Sets, Multisets, and Partitions</b>	<b>349</b>
12.1 Basics . . . . .	350
12.2 Array and Bit-Vector Sets . . . . .	351
12.3 Multisets . . . . .	359
12.4 Partitions . . . . .	364
12.5 Applications . . . . .	376
Exercises . . . . .	378
Programming Projects . . . . .	379
<b>13 Garbage Collection</b>	<b>381</b>
13.1 What is Garbage? . . . . .	381
13.2 Reference Counting Garbage Collection . . . . .	384
13.3 Mark-and-Sweep Garbage Collection . . . . .	388
13.4 Stop-and-Copy Garbage Collection . . . . .	391
13.5 Mark-and-Compact Garbage Collection . . . . .	392
Exercises . . . . .	395
Programming Projects . . . . .	396
<b>14 Algorithmic Patterns and Problem Solvers</b>	<b>399</b>
14.1 Brute-Force and Greedy Algorithms . . . . .	399
14.2 Backtracking Algorithms . . . . .	402
14.3 Top-Down Algorithms: Divide-and-Conquer . . . . .	410
14.4 Bottom-Up Algorithms: Dynamic Programming . . . . .	419
14.5 Randomized Algorithms . . . . .	427
Exercises . . . . .	436
Programming Projects . . . . .	438
<b>15 Sorting Algorithms and Sorters</b>	<b>441</b>
15.1 Basics . . . . .	441
15.2 Sorting and Sorters . . . . .	442
15.3 Insertion Sorting . . . . .	444
15.4 Exchange Sorting . . . . .	448
15.5 Selection Sorting . . . . .	459
15.6 Merge Sorting . . . . .	467
15.7 A Lower Bound on Sorting . . . . .	472
15.8 Distribution Sorting . . . . .	473
15.9 Performance Data . . . . .	479
Exercises . . . . .	481
Programming Projects . . . . .	484

---

<b>16 Graphs and Graph Algorithms</b>	<b>485</b>
16.1 Basics . . . . .	486
16.2 Implementing Graphs . . . . .	493
16.3 Graph Traversals . . . . .	503
16.4 Shortest-Path Algorithms . . . . .	515
16.5 Minimum-Cost Spanning Trees . . . . .	523
16.6 Application: Critical Path Analysis . . . . .	533
Exercises . . . . .	537
Programming Projects . . . . .	540
<b>A Ruby and Object-Oriented Programming</b>	<b>543</b>
A.1 Objects and Types . . . . .	543
A.2 Variables . . . . .	543
A.3 Parameter Passing . . . . .	545
A.4 Classes . . . . .	546
A.5 Nested Classes . . . . .	550
A.6 Inheritance and Polymorphism . . . . .	551
A.7 Exceptions . . . . .	557
<b>B Class Hierarchy Diagrams</b>	<b>559</b>
<b>C Character Codes</b>	<b>561</b>
<b>Index</b>	<b>567</b>

# Preface

This book was motivated by my experience in teaching the course *EECE 250: Algorithms and Data Structures* in the Computer Engineering program at the University of Waterloo. I have observed that the advent of *object-oriented methods* and the emergence of object-oriented *design patterns* has led to a profound change in the pedagogy of data structures and algorithms. The successful application of these techniques gives rise to a kind of cognitive unification: Ideas that are disparate and apparently unrelated seem to come together when the appropriate design patterns and abstractions are used.

This paradigm shift is both evolutionary and revolutionary. On the one hand, the knowledge base grows incrementally as programmers and researchers invent new algorithms and data structures. On the other hand, the proper use of object-oriented techniques requires a fundamental change in the way the programs are designed and implemented. Programmers who are well schooled in the procedural ways often find the leap to objects to be a difficult one.

## ❖ Goals

The primary goal of this book is to promote object-oriented design using Ruby and to illustrate the use of the emerging *object-oriented design patterns*. Experienced object-oriented programmers find that certain ways of doing things work best and that these ways occur over and over again. The book shows how these patterns are used to create good software designs. In particular, the following design patterns are used throughout the text: *singleton*, *container*, *iterator*, *adapter* and *visitor*.

Virtually all of the data structures are presented in the context of a *single, unified, polymorphic class hierarchy*. This framework clearly shows the *relationships* between data structures and it illustrates how polymorphism and inheritance can be used effectively. In addition, *algorithmic abstraction* is used extensively when presenting classes of algorithms. By using algorithmic abstraction, it is possible to describe a generic algorithm without having to worry about the details of a particular concrete realization of that algorithm.

A secondary goal of the book is to present mathematical tools *just in time*. Analysis techniques and proofs are presented as needed and in the proper context. In the past when the topics in this book were taught at the graduate level, an author could rely on students having the needed background in mathematics. However, because the book is targeted for second- and third-year students, it is necessary to fill in the background as needed. To the extent possible without compromising correctness, the presentation fosters intuitive understanding of the concepts rather than mathematical rigor.

## ❖ Approach

One cannot learn to program just by reading a book. It is a skill that must be developed by practice. Nevertheless, the best practitioners study the works of others and incorporate their observations into their own practice. I firmly believe that after learning the rudiments of program writing, students should be exposed to examples of complex, yet well-designed program artifacts so that they can learn about the designing good software.

Consequently, this book presents the various data structures and algorithms as complete Ruby program fragments. All the program fragments presented in this book have been extracted automatically from the source code files of working and tested programs. It has been my experience that by developing the proper abstractions, it is possible to present the concepts as fully functional programs without resorting to *pseudo-code* or to hand-waving.

## ❖ Outline

This book presents material identified in the *Computing Curricula 1991* report of the ACM/IEEE-CS Joint Curriculum Task Force[49]. The book specifically addresses the following *knowledge units*: AL1: Basic Data structures, AL2: Abstract Data Types, AL3: Recursive Algorithms, AL4: Complexity Analysis, AL6: Sorting and Searching, and AL8: Problem-Solving Strategies. The breadth and depth of coverage is typical of what should appear in the second or third year of an undergraduate program in computer science/computer engineering.

In order to analyze a program, it is necessary to develop a model of the computer. Chapter 2 develops several models and illustrates with examples how these models predict performance. Both average-case and worst-case analyses of running time are considered. Recursive algorithms are discussed and it is shown how to solve a recurrence using repeated substitution. This chapter also reviews arithmetic and geometric series summations, Horner's rule and the properties of harmonic numbers.

Chapter 3 introduces asymptotic (big-oh) notation and shows by comparing with Chapter 2 that the results of asymptotic analysis are consistent with models of higher fidelity. In addition to  $O(\cdot)$ , this chapter also covers other asymptotic notations ( $\Omega(\cdot)$ ,  $\Theta(\cdot)$ , and  $o(\cdot)$ ) and develops the asymptotic properties of polynomials and logarithms.

Chapter 4 introduces the *foundational data structures*—the array and the linked list. Virtually all the data structures in the rest of the book can be implemented using either one of these foundational structures. This chapter also covers multi-dimensional arrays and matrices.

Chapter 5 deals with abstraction and data types. It presents the recurring design patterns used throughout the text as well a unifying framework for the data structures presented in the subsequent chapters. In particular, all of the data structures are viewed as *abstract containers*.

Chapter 6 discusses stacks, queues, and deques. This chapter presents implementations based on both foundational data structures (arrays and linked lists). Applications for stacks and queues are presented.

Chapter 7 covers ordered lists, both sorted and unsorted. In this chapter, a list is viewed as a *searchable container*. Again several applications of lists are presented.

Chapter 8 introduces hashing and the notion of a hash table. This chapter addresses the design of hashing functions for the various basic data types as well as

for the abstract data types described in Chapter 5. Both scatter tables and hash tables are covered in depth and analytical performance results are derived.

Chapter 9 introduces trees and describes their many forms. Both depth-first and breadth-first tree traversals are presented. Completely generic traversal algorithms based on the use of the *visitor* design pattern are presented, thereby illustrating the power of *algorithmic abstraction*. This chapter also shows how trees are used to represent mathematical expressions and illustrates the relationships between traversals and the various expression notations (prefix, infix, and postfix).

Chapter 10 addresses trees as *searchable containers*. Again, the power of *algorithmic abstraction* is demonstrated by showing the relationships between simple algorithms and balancing algorithms. This chapter also presents average case performance analyses and illustrates the solution of recurrences by telescoping.

Chapter 11 presents several priority queue implementations, including binary heaps, leftist heaps, and binomial queues. In particular this chapter illustrates how a more complicated data structure (leftist heap) extends an existing one (tree). Discrete-event simulation is presented as an application of priority queues.

Chapter 12 covers sets and multisets. Also covered are partitions and disjoint set algorithms. The latter topic illustrates again the use of algorithmic abstraction.

Garbage collection is discussed in Chapter 13. This is a topic that is not found often in texts of this sort. However, because the Ruby language relies on garbage collection, it is important to understand how it works and how it affects the running times of programs.

Chapter 14 surveys a number of algorithm design techniques. Included are brute-force and greedy algorithms, backtracking algorithms (including branch-and-bound), divide-and-conquer algorithms, and dynamic programming. An object-oriented approach based on the notion of an *abstract solution space* and an *abstract solver* unifies much of the discussion. This chapter also covers briefly random number generators, Monte Carlo methods, and simulated annealing.

Chapter 15 covers the major sorting algorithms in an object-oriented style based on the notion of an *abstract sorter*. Using the abstract sorter illustrates the relationships between the various classes of sorting algorithm and demonstrates the use of algorithmic abstractions.

Finally, Chapter 16 presents an overview of graphs and graph algorithms. Both depth-first and breadth-first graph traversals are presented. Topological sort is viewed as yet another special kind of traversal. Generic traversal algorithms based on the *visitor* design pattern are presented, once more illustrating *algorithmic abstraction*. This chapter also covers various shortest path algorithms and minimum-spanning-tree algorithms.

At the end of each chapter is a set of exercises and a set of programming projects. The exercises are designed to consolidate the concepts presented in the text. The programming projects generally require the student to extend the implementation given in the text.

## ❖ Suggested Course Outline

This text may be used in either a one semester or a two semester course. The course which I teach at Waterloo is a one-semester course that comprises 36 lecture hours on the following topics:

1. Review of the fundamentals of programming in Ruby and an overview of object-oriented programming with Ruby. (Appendix A). [4 lecture hours].
2. Models of the computer, algorithm analysis, and asymptotic notation (Chapters 2 and 3). [4 lecture hours].
3. Foundational data structures, abstraction, and abstract data types (Chapters 4 and 5). [4 lecture hours].
4. Stacks, queues, ordered lists, and sorted lists (Chapters 6 and 7). [3 lecture hours].
5. Hashing, hash tables, and scatter tables (Chapter 8). [3 lecture hours].
6. Trees and search trees (Chapters 9 and 10). [6 lecture hours].
7. Heaps and priority queues (Chapter 11). [3 lecture hours].
8. Algorithm design techniques (Chapter 14). [3 lecture hours].
9. Sorting algorithms and sorters (Chapter 15). [3 lecture hours].
10. Graphs and graph algorithms (Chapter 16). [3 lecture hours].

Depending on the background of students, a course instructor may find it necessary to review features of the Ruby language. For example, students need to understand the use of the `yield` statement to define a Ruby *generator* method and how the Ruby `for` statement makes use of programmer-defined *generators*. Similarly, an understanding of the workings of Ruby *modules*, *classes* and *inheritance*, is required in order to understand the unifying class hierarchy discussed in Chapter 5.

### ❖ Online Course Materials

Additional material supporting this book can be found on the world-wide web at the URL:

<http://www.brpreiss.com/books/opus8>

In particular, you will find there the source code for all the program fragments in this book as well as an errata list.



### Acknowledgments

Insert acknowledgements here.

Waterloo, Canada  
November 12, 2004