

Open
Data
Structures

An
Introduction

PAT
MORIN



Open Data Structures

OPEL (OPEN PATHS TO ENRICHED LEARNING)

Series Editor: Connor Houlihan

Open Paths to Enriched Learning (OPEL) reflects the continued commitment of Athabasca University to removing barriers — including the cost of course materials — that restrict access to university-level study. The OPEL series offers introductory texts, on a broad array of topics, written especially with undergraduate students in mind. Although the books in the series are designed for course use, they also afford lifelong learners an opportunity to enrich their own knowledge. Like all AU Press publications, OPEL course texts are available for free download at www.aupress.ca, as well as for purchase in both print and digital formats.

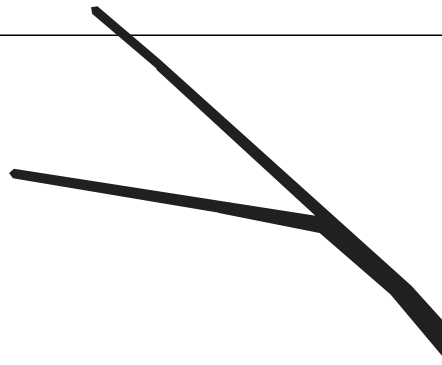
SERIES TITLES

Open Data Structures: An Introduction
Pat Morin



Athabasca
University

Open
Data
Structures



An
Introduction

PAT
MORIN



Copyright © 2013 Pat Morin

Published by AU Press, Athabasca University
1200, 10011-109 Street, Edmonton, AB T5J 3S8

A volume in OPEL (Open Paths to Enriched Learning)
ISSN 2291-2606 (print) 2291-2614 (digital)

Cover and interior design by Marvin Harder, marvinharder.com.
Printed and bound in Canada by Marquis Book Printers.

Library and Archives Canada Cataloguing in Publication
Morin, Pat, 1973—, author
Open data structures : an introduction / Pat Morin.

(OPEL (Open paths to enriched learning), ISSN 2291-2606 ; 1)
Includes bibliographical references and index.
Issued in print and electronic formats.
ISBN 978-1-927356-38-8 (pbk.)—ISBN 978-1-927356-39-5 (pdf).—
ISBN 978-1-927356-40-1 (epub)

1. Data structures (Computer science). 2. Computer algorithms.
I. Title. II. Series: Open paths to enriched learning ; 1

QA76.9.D35M67 2013 005.7'3 C2013-902170-1

We acknowledge the financial support of the Government of Canada through the Canada Book Fund (CBF) for our publishing activities.



Assistance provided by the Government of Alberta, Alberta Multimedia Development Fund.



This publication is licensed under a Creative Commons license, Attribution-Noncommercial-No Derivative Works 2.5 Canada: see www.creativecommons.org. The text may be reproduced for non-commercial purposes, provided that credit is given to the original author.

To obtain permission for uses beyond those outlined in the Creative Commons license, please contact AU Press, Athabasca University, at aupress@athabascau.ca.

Contents

Acknowledgments	xi
Why This Book?	xiii
1 Introduction	1
1.1 The Need for Efficiency	2
1.2 Interfaces	4
1.2.1 The Queue, Stack, and Deque Interfaces	5
1.2.2 The List Interface: Linear Sequences	6
1.2.3 The USet Interface: Unordered Sets	8
1.2.4 The SSet Interface: Sorted Sets	9
1.3 Mathematical Background	9
1.3.1 Exponentials and Logarithms	10
1.3.2 Factorials	11
1.3.3 Asymptotic Notation	12
1.3.4 Randomization and Probability	15
1.4 The Model of Computation	18
1.5 Correctness, Time Complexity, and Space Complexity	19
1.6 Code Samples	22
1.7 List of Data Structures	22
1.8 Discussion and Exercises	26
2 Array-Based Lists	29
2.1 ArrayStack: Fast Stack Operations Using an Array	30
2.1.1 The Basics	30
2.1.2 Growing and Shrinking	33
2.1.3 Summary	35

2.2	FastArrayStack: An Optimized ArrayStack	35
2.3	ArrayQueue: An Array-Based Queue	36
2.3.1	Summary	40
2.4	ArrayDeque: Fast Deque Operations Using an Array	40
2.4.1	Summary	43
2.5	DualArrayDeque: Building a Deque from Two Stacks	43
2.5.1	Balancing	47
2.5.2	Summary	49
2.6	RootishArrayStack: A Space-Efficient Array Stack	49
2.6.1	Analysis of Growing and Shrinking	54
2.6.2	Space Usage	54
2.6.3	Summary	55
2.6.4	Computing Square Roots	56
2.7	Discussion and Exercises	59
3	Linked Lists	63
3.1	SLList: A Singly-Linked List	63
3.1.1	Queue Operations	65
3.1.2	Summary	66
3.2	DLList: A Doubly-Linked List	67
3.2.1	Adding and Removing	69
3.2.2	Summary	70
3.3	SEList: A Space-Efficient Linked List	71
3.3.1	Space Requirements	72
3.3.2	Finding Elements	73
3.3.3	Adding an Element	74
3.3.4	Removing an Element	77
3.3.5	Amortized Analysis of Spreading and Gathering	79
3.3.6	Summary	81
3.4	Discussion and Exercises	82
4	Skiplists	87
4.1	The Basic Structure	87
4.2	SkiplistSSet: An Efficient SSet	90
4.2.1	Summary	93
4.3	SkiplistList: An Efficient Random-Access List	93

4.3.1	Summary	98
4.4	Analysis of Skiplists	98
4.5	Discussion and Exercises	102
5	Hash Tables	107
5.1	ChainedHashTable: Hashing with Chaining	107
5.1.1	Multiplicative Hashing	110
5.1.2	Summary	114
5.2	LinearHashTable: Linear Probing	114
5.2.1	Analysis of Linear Probing	118
5.2.2	Summary	121
5.2.3	Tabulation Hashing	121
5.3	Hash Codes	122
5.3.1	Hash Codes for Primitive Data Types	123
5.3.2	Hash Codes for Compound Objects	123
5.3.3	Hash Codes for Arrays and Strings	125
5.4	Discussion and Exercises	128
6	Binary Trees	133
6.1	BinaryTree: A Basic Binary Tree	135
6.1.1	Recursive Algorithms	136
6.1.2	Traversing Binary Trees	136
6.2	BinarySearchTree: An Unbalanced Binary Search Tree	140
6.2.1	Searching	140
6.2.2	Addition	142
6.2.3	Removal	144
6.2.4	Summary	146
6.3	Discussion and Exercises	147
7	Random Binary Search Trees	153
7.1	Random Binary Search Trees	153
7.1.1	Proof of Lemma 7.1	156
7.1.2	Summary	158
7.2	Treap: A Randomized Binary Search Tree	159
7.2.1	Summary	166
7.3	Discussion and Exercises	168

8	Scapegoat Trees	173
8.1	ScapegoatTree: A Binary Search Tree with Partial Rebuilding	174
8.1.1	Analysis of Correctness and Running-Time	178
8.1.2	Summary	180
8.2	Discussion and Exercises	181
9	Red-Black Trees	185
9.1	2-4 Trees	186
9.1.1	Adding a Leaf	187
9.1.2	Removing a Leaf	187
9.2	RedBlackTree: A Simulated 2-4 Tree	190
9.2.1	Red-Black Trees and 2-4 Trees	190
9.2.2	Left-Leaning Red-Black Trees	194
9.2.3	Addition	196
9.2.4	Removal	199
9.3	Summary	205
9.4	Discussion and Exercises	206
10	Heaps	211
10.1	BinaryHeap: An Implicit Binary Tree	211
10.1.1	Summary	217
10.2	MeldableHeap: A Randomized Meldable Heap	217
10.2.1	Analysis of merge(h1,h2)	220
10.2.2	Summary	221
10.3	Discussion and Exercises	222
11	Sorting Algorithms	225
11.1	Comparison-Based Sorting	226
11.1.1	Merge-Sort	226
11.1.2	Quicksort	230
11.1.3	Heap-sort	233
11.1.4	A Lower-Bound for Comparison-Based Sorting	235
11.2	Counting Sort and Radix Sort	238
11.2.1	Counting Sort	239
11.2.2	Radix-Sort	241

11.3 Discussion and Exercises	243
12 Graphs	247
12.1 AdjacencyMatrix: Representing a Graph by a Matrix	249
12.2 AdjacencyLists: A Graph as a Collection of Lists	252
12.3 Graph Traversal	256
12.3.1 Breadth-First Search	256
12.3.2 Depth-First Search	258
12.4 Discussion and Exercises	261
13 Data Structures for Integers	265
13.1 BinaryTrie: A digital search tree	266
13.2 XFastTrie: Searching in Doubly-Logarithmic Time	272
13.3 YFastTrie: A Doubly-Logarithmic Time SSet	275
13.4 Discussion and Exercises	280
14 External Memory Searching	283
14.1 The Block Store	285
14.2 B-Trees	285
14.2.1 Searching	288
14.2.2 Addition	290
14.2.3 Removal	295
14.2.4 Amortized Analysis of <i>B</i> -Trees	301
14.3 Discussion and Exercises	304
Bibliography	309
Index	317

Acknowledgments

I am grateful to Nima Hoda, who spent a summer tirelessly proofreading many of the chapters in this book; to the students in the Fall 2011 offering of COMP2402/2002, who put up with the first draft of this book and spotted many typographic, grammatical, and factual errors; and to Morgan Tunzelmann at Athabasca University Press, for patiently editing several near-final drafts.

Why This Book?

There are plenty of books that teach introductory data structures. Some of them are very good. Most of them cost money, and the vast majority of computer science undergraduate students will shell out at least some cash on a data structures book.

Several free data structures books are available online. Some are very good, but most of them are getting old. The majority of these books became free when their authors and/or publishers decided to stop updating them. Updating these books is usually not possible, for two reasons: (1) The copyright belongs to the author and/or publisher, either of whom may not allow it. (2) The *source code* for these books is often not available. That is, the Word, WordPerfect, FrameMaker, or \LaTeX source for the book is not available, and even the version of the software that handles this source may not be available.

The goal of this project is to free undergraduate computer science students from having to pay for an introductory data structures book. I have decided to implement this goal by treating this book like an Open Source software project. The \LaTeX source, Java source, and build scripts for the book are available to download from the author's website¹ and also, more importantly, on a reliable source code management site.²

The source code available there is released under a Creative Commons Attribution license, meaning that anyone is free to *share*: to copy, distribute and transmit the work; and to *remix*: to adapt the work, including the right to make commercial use of the work. The only condition on these rights is *attribution*: you must acknowledge that the derived work contains code and/or text from opendatastructures.org.

¹<http://opendatastructures.org>

²<https://github.com/patmorin/ods>

Anyone can contribute corrections/fixes using the `git` source-code management system. Anyone can also fork the book's sources to develop a separate version (for example, in another programming language). My hope is that, by doing things this way, this book will continue to be a useful textbook long after my interest in the project or my pulse, (whichever comes first) has waned.

Chapter 1

Introduction

Every computer science curriculum in the world includes a course on data structures and algorithms. Data structures are *that* important; they improve our quality of life and even save lives on a regular basis. Many multi-million and several multi-billion dollar companies have been built around data structures.

How can this be? If we stop to think about it, we realize that we interact with data structures constantly.

- Open a file: File system data structures are used to locate the parts of that file on disk so they can be retrieved. This isn't easy; disks contain hundreds of millions of blocks. The contents of your file could be stored on any one of them.
- Look up a contact on your phone: A data structure is used to look up a phone number in your contact list based on partial information even before you finish dialing/typing. This isn't easy; your phone may contain information about a lot of people—everyone you have ever contacted via phone or email—and your phone doesn't have a very fast processor or a lot of memory.
- Log in to your favourite social network: The network servers use your login information to look up your account information. This isn't easy; the most popular social networks have hundreds of millions of active users.
- Do a web search: The search engine uses data structures to find the web pages containing your search terms. This isn't easy; there are

over 8.5 billion web pages on the Internet and each page contains a lot of potential search terms.

- Phone emergency services (9-1-1): The emergency services network looks up your phone number in a data structure that maps phone numbers to addresses so that police cars, ambulances, or fire trucks can be sent there without delay. This is important; the person making the call may not be able to provide the exact address they are calling from and a delay can mean the difference between life or death.

1.1 The Need for Efficiency

In the next section, we look at the operations supported by the most commonly used data structures. Anyone with a bit of programming experience will see that these operations are not hard to implement correctly. We can store the data in an array or a linked list and each operation can be implemented by iterating over all the elements of the array or list and possibly adding or removing an element.

This kind of implementation is easy, but not very efficient. Does this really matter? Computers are becoming faster and faster. Maybe the obvious implementation is good enough. Let's do some rough calculations to find out.

Number of operations: Imagine an application with a moderately-sized data set, say of one million (10^6), items. It is reasonable, in most applications, to assume that the application will want to look up each item at least once. This means we can expect to do at least one million (10^6) searches in this data. If each of these 10^6 searches inspects each of the 10^6 items, this gives a total of $10^6 \times 10^6 = 10^{12}$ (one thousand billion) inspections.

Processor speeds: At the time of writing, even a very fast desktop computer can not do more than one billion (10^9) operations per second.¹ This

¹Computer speeds are at most a few gigahertz (billions of cycles per second), and each operation typically takes a few cycles.

means that this application will take at least $10^{12}/10^9 = 1000$ seconds, or roughly 16 minutes and 40 seconds. Sixteen minutes is an eon in computer time, but a person might be willing to put up with it (if he or she were headed out for a coffee break).

Bigger data sets: Now consider a company like Google, that indexes over 8.5 billion web pages. By our calculations, doing any kind of query over this data would take at least 8.5 seconds. We already know that this isn't the case; web searches complete in much less than 8.5 seconds, and they do much more complicated queries than just asking if a particular page is in their list of indexed pages. At the time of writing, Google receives approximately 4,500 queries per second, meaning that they would require at least $4,500 \times 8.5 = 38,250$ very fast servers just to keep up.

The solution: These examples tell us that the obvious implementations of data structures do not scale well when the number of items, n , in the data structure and the number of operations, m , performed on the data structure are both large. In these cases, the time (measured in, say, machine instructions) is roughly $n \times m$.

The solution, of course, is to carefully organize data within the data structure so that not every operation requires every data item to be inspected. Although it sounds impossible at first, we will see data structures where a search requires looking at only two items on average, independent of the number of items stored in the data structure. In our billion instruction per second computer it takes only 0.000000002 seconds to search in a data structure containing a billion items (or a trillion, or a quadrillion, or even a quintillion items).

We will also see implementations of data structures that keep the items in sorted order, where the number of items inspected during an operation grows very slowly as a function of the number of items in the data structure. For example, we can maintain a sorted set of one billion items while inspecting at most 60 items during any operation. In our billion instruction per second computer, these operations take 0.00000006 seconds each.

The remainder of this chapter briefly reviews some of the main concepts used throughout the rest of the book. Section 1.2 describes the in-

terfaces implemented by all of the data structures described in this book and should be considered required reading. The remaining sections discuss:

- some mathematical review including exponentials, logarithms, factorials, asymptotic (big-Oh) notation, probability, and randomization;
- the model of computation;
- correctness, running time, and space;
- an overview of the rest of the chapters; and
- the sample code and typesetting conventions.

A reader with or without a background in these areas can easily skip them now and come back to them later if necessary.

1.2 Interfaces

When discussing data structures, it is important to understand the difference between a data structure's interface and its implementation. An interface describes what a data structure does, while an implementation describes how the data structure does it.

An *interface*, sometimes also called an *abstract data type*, defines the set of operations supported by a data structure and the semantics, or meaning, of those operations. An interface tells us nothing about how the data structure implements these operations; it only provides a list of supported operations along with specifications about what types of arguments each operation accepts and the value returned by each operation.

A data structure *implementation*, on the other hand, includes the internal representation of the data structure as well as the definitions of the algorithms that implement the operations supported by the data structure. Thus, there can be many implementations of a single interface. For example, in Chapter 2, we will see implementations of the `List` interface using arrays and in Chapter 3 we will see implementations of the `List` interface using pointer-based data structures. Each implements the same interface, `List`, but in different ways.

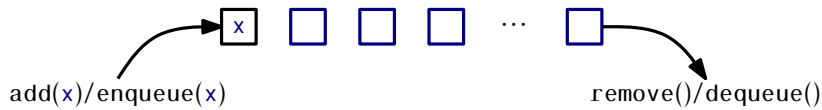


Figure 1.1: A FIFO Queue.

1.2.1 The Queue, Stack, and Deque Interfaces

The Queue interface represents a collection of elements to which we can add elements and remove the next element. More precisely, the operations supported by the Queue interface are

- `add(x)`: add the value `x` to the Queue
- `remove()`: remove the next (previously added) value, `y`, from the Queue and return `y`

Notice that the `remove()` operation takes no argument. The Queue's *queueing discipline* decides which element should be removed. There are many possible queueing disciplines, the most common of which include FIFO, priority, and LIFO.

A *FIFO (first-in-first-out) Queue*, which is illustrated in Figure 1.1, removes items in the same order they were added, much in the same way a queue (or line-up) works when checking out at a cash register in a grocery store. This is the most common kind of Queue so the qualifier FIFO is often omitted. In other texts, the `add(x)` and `remove()` operations on a FIFO Queue are often called `enqueue(x)` and `dequeue()`, respectively.

A *priority Queue*, illustrated in Figure 1.2, always removes the smallest element from the Queue, breaking ties arbitrarily. This is similar to the way in which patients are triaged in a hospital emergency room. As patients arrive they are evaluated and then placed in a waiting room. When a doctor becomes available he or she first treats the patient with the most life-threatening condition. The `remove(x)` operation on a priority Queue is usually called `deleteMin()` in other texts.

A very common queueing discipline is the LIFO (last-in-first-out) discipline, illustrated in Figure 1.3. In a *LIFO Queue*, the most recently added element is the next one removed. This is best visualized in terms of a stack of plates; plates are placed on the top of the stack and also

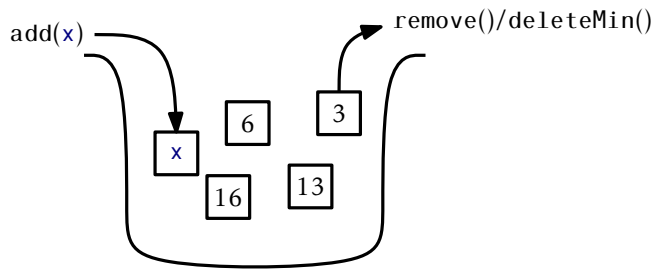


Figure 1.2: A priority Queue.

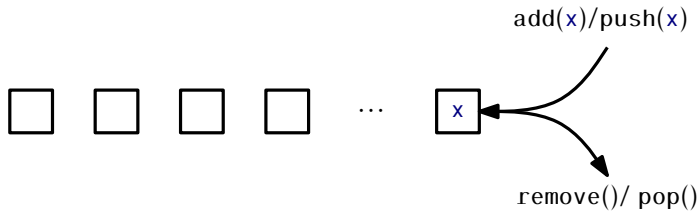


Figure 1.3: A stack.

removed from the top of the stack. This structure is so common that it gets its own name: Stack. Often, when discussing a Stack, the names of `add(x)` and `remove()` are changed to `push(x)` and `pop()`; this is to avoid confusing the LIFO and FIFO queueing disciplines.

A Deque is a generalization of both the FIFO Queue and LIFO Queue (Stack). A Deque represents a sequence of elements, with a front and a back. Elements can be added at the front of the sequence or the back of the sequence. The names of the Deque operations are self-explanatory: `addFirst(x)`, `removeFirst()`, `addLast(x)`, and `removeLast()`. It is worth noting that a Stack can be implemented using only `addFirst(x)` and `removeFirst()` while a FIFO Queue can be implemented using `addLast(x)` and `removeFirst()`.

1.2.2 The List Interface: Linear Sequences

This book will talk very little about the FIFO Queue, Stack, or Deque interfaces. This is because these interfaces are subsumed by the List interface. A List, illustrated in Figure 1.4, represents a sequence, x_0, \dots, x_{n-1} ,

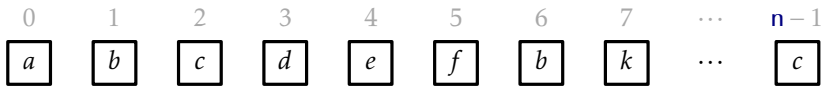


Figure 1.4: A List represents a sequence indexed by $0, 1, 2, \dots, n$. In this List a call to `get(2)` would return the value `c`.

of values. The List interface includes the following operations:

1. `size()`: return n , the length of the list
2. `get(i)`: return the value x_i
3. `set(i, x)`: set the value of x_i equal to x
4. `add(i, x)`: add x at position i , displacing x_i, \dots, x_{n-1} ;
Set $x_{j+1} = x_j$, for all $j \in \{n-1, \dots, i\}$, increment n , and set $x_i = x$
5. `remove(i)` remove the value x_i , displacing x_{i+1}, \dots, x_{n-1} ;
Set $x_j = x_{j+1}$, for all $j \in \{i, \dots, n-2\}$ and decrement n

Notice that these operations are easily sufficient to implement the Deque interface:

$$\begin{aligned} \text{addFirst}(x) &\Rightarrow \text{add}(0, x) \\ \text{removeFirst}() &\Rightarrow \text{remove}(0) \\ \text{addLast}(x) &\Rightarrow \text{add}(\text{size}(), x) \\ \text{removeLast}() &\Rightarrow \text{remove}(\text{size}() - 1) \end{aligned}$$

Although we will normally not discuss the Stack, Deque and FIFO Queue interfaces in subsequent chapters, the terms Stack and Deque are sometimes used in the names of data structures that implement the List interface. When this happens, it highlights the fact that these data structures can be used to implement the Stack or Deque interface very efficiently. For example, the `ArrayDeque` class is an implementation of the List interface that implements all the Deque operations in constant time per operation.

1.2.3 The USet Interface: Unordered Sets

The USet interface represents an unordered set of unique elements, which mimics a mathematical *set*. A USet contains *n* *distinct* elements; no element appears more than once; the elements are in no specific order. A USet supports the following operations:

1. `size()`: return the number, *n*, of elements in the set
2. `add(x)`: add the element *x* to the set if not already present; Add *x* to the set provided that there is no element *y* in the set such that *x* equals *y*. Return `true` if *x* was added to the set and `false` otherwise.
3. `remove(x)`: remove *x* from the set; Find an element *y* in the set such that *x* equals *y* and remove *y*. Return *y*, or `null` if no such element exists.
4. `find(x)`: find *x* in the set if it exists; Find an element *y* in the set such that *y* equals *x*. Return *y*, or `null` if no such element exists.

These definitions are a bit fussy about distinguishing *x*, the element we are removing or finding, from *y*, the element we may remove or find. This is because *x* and *y* might actually be distinct objects that are nevertheless treated as equal.² Such a distinction is useful because it allows for the creation of *dictionaries* or *maps* that map keys onto values.

To create a dictionary/map, one forms compound objects called `Pairs`, each of which contains a *key* and a *value*. Two `Pairs` are treated as equal if their keys are equal. If we store some pair (k, v) in a USet and then later call the `find(x)` method using the pair $x = (k, null)$ the result will be $y = (k, v)$. In other words, it is possible to recover the value, *v*, given only the key, *k*.

²In Java, this is done by overriding the class's `equals(y)` and `hashCode()` methods.

1.2.4 The SSet Interface: Sorted Sets

The SSet interface represents a sorted set of elements. An SSet stores elements from some total order, so that any two elements x and y can be compared. In code examples, this will be done with a method called `compare(x, y)` in which

$$\text{compare}(x, y) \begin{cases} < 0 & \text{if } x < y \\ > 0 & \text{if } x > y \\ = 0 & \text{if } x = y \end{cases}$$

An SSet supports the `size()`, `add(x)`, and `remove(x)` methods with exactly the same semantics as in the USet interface. The difference between a USet and an SSet is in the `find(x)` method:

4. `find(x)`: locate x in the sorted set;
Find the smallest element y in the set such that $y \geq x$. Return y or `null` if no such element exists.

This version of the `find(x)` operation is sometimes referred to as a *successor search*. It differs in a fundamental way from `USet.find(x)` since it returns a meaningful result even when there is no element equal to x in the set.

The distinction between the USet and SSet `find(x)` operations is very important and often missed. The extra functionality provided by an SSet usually comes with a price that includes both a larger running time and a higher implementation complexity. For example, most of the SSet implementations discussed in this book all have `find(x)` operations with running times that are logarithmic in the size of the set. On the other hand, the implementation of a USet as a `ChainedHashTable` in Chapter 5 has a `find(x)` operation that runs in constant expected time. When choosing which of these structures to use, one should always use a USet unless the extra functionality offered by an SSet is truly needed.

1.3 Mathematical Background

In this section, we review some mathematical notations and tools used throughout this book, including logarithms, big-Oh notation, and proba-

- [download Red Army Tank Commander: At War in a T-34 on the Eastern Front](#)
- [read online The Gargoyle King \(Dragonlance: Ogre Titans, Book 3\)](#)
- [Non-Essential Mnemonics: An Unnecessary Journey into Senseless Knowledge pdf](#)
- [Work's New Age: The End of Full Employment and What It Means to You online](#)

- <http://growingsomeroots.com/ebooks/Meditate-Your-Weight--A-21-Day-Retreat-to-Optimize-Your-Metabolism-and-Feel-Great.pdf>
- <http://dadhoc.com/lib/Technology-as-Human-Social-Tradition--Cultural-Transmission-among-Hunter-Gatherers--Origins-of-Human-Behavior-and-C>
- <http://transtrade.cz/?ebooks/British-Broadcasting--Radio-and-Television-in-the-United-Kingdom.pdf>
- <http://thermco.pl/library/Games-World-of-Puzzles--May-2016-.pdf>