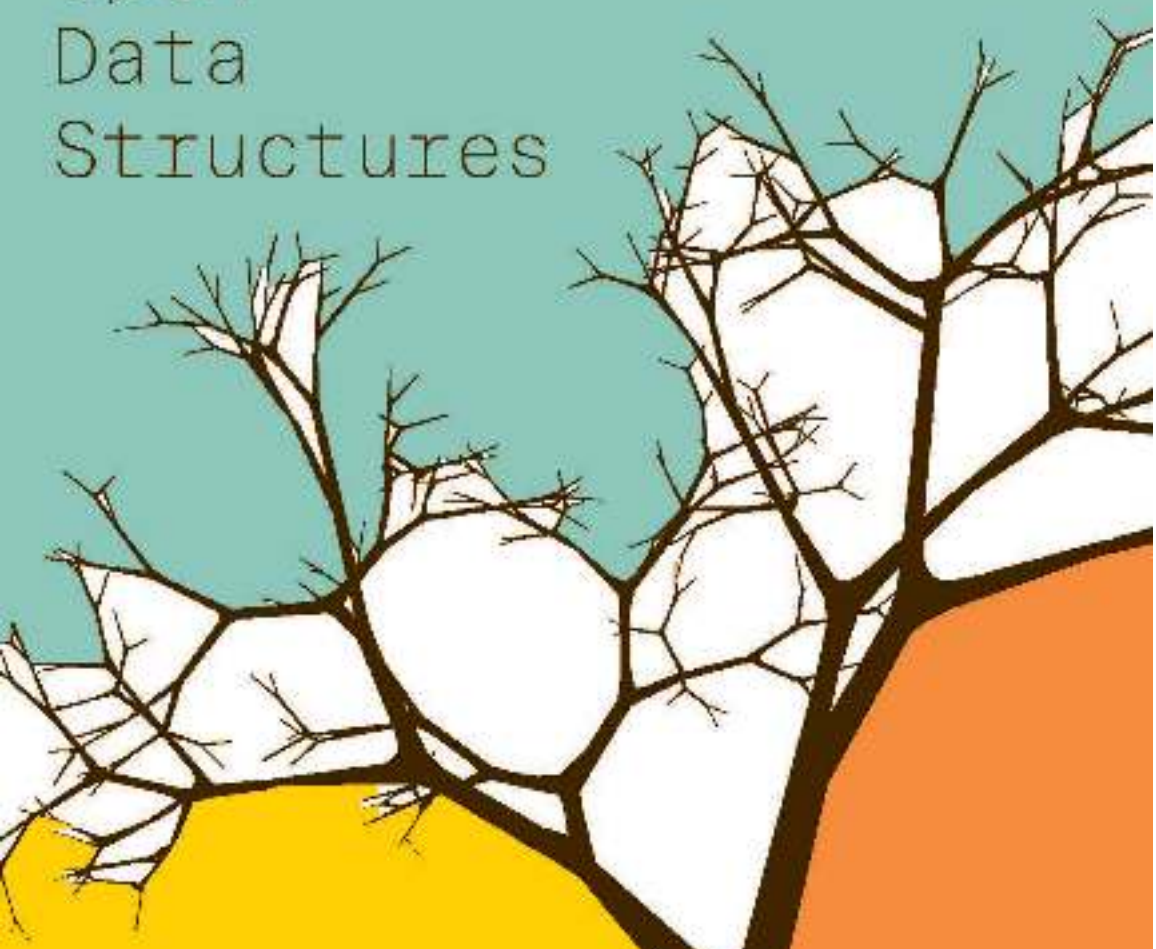
Open Data Structures



An Introduction

PAT MORIN

OPEL (OPEN PATHS TO ENRICHED LEARNING)

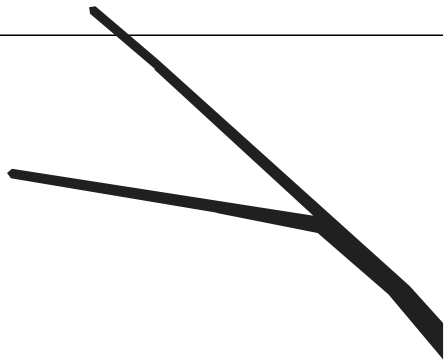
Series Editor: Connor Houlihan

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

Open Data Structures



An Introduction

PAT MORIN

AU PRESS

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.

Canadian Heritage Patrimoine canadien

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

Alberta Government

Contents

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 website[1] 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

[1] Computer 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.

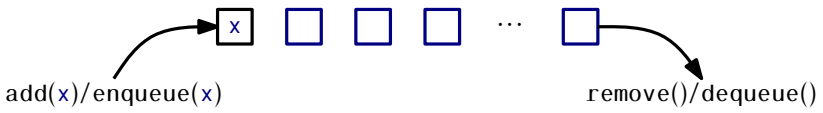


Figure 1.1: A FIFO Queue.

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

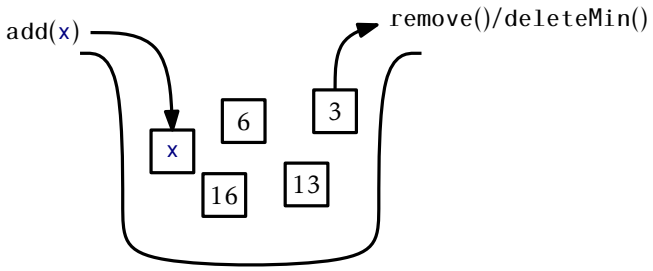


Figure 1.2: A priority Queue.

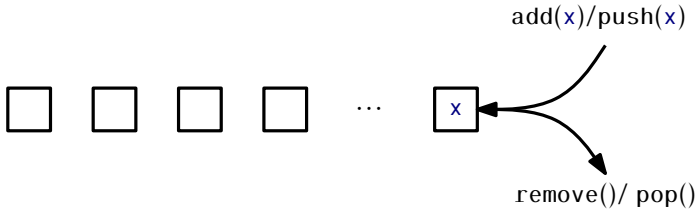


Figure 1.3: A stack.

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, $x_0,\ldots,x_{n-1}$,

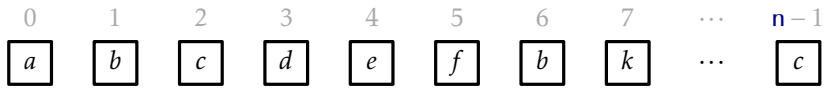


Figure 1.4: A `List` represents a sequence indexed by $0, 1, 2, \ldots, n$. In this `List` a call to `get(2)` would return the value c.

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 $x_i, \ldots, x_{n-1}$;
   Set $x_{j+1} = x_j$, for all $j \in \{n-1, \ldots, i\}$, increment n, and set $x_i = x$
5. `remove(i)` remove the value x_i, displacing $x_{i+1}, \ldots, x_{n-1}$;
   Set $x_j = x_{j+1}$, for all $j \in \{i, \ldots, n-2\}$ and decrement n

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

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

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.

[2] In 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

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

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-