Advanced Data Structures

Advanced Data Structures presents a comprehensive look at the ideas, analysis, and implementation details of data structures as a specialized topic in applied algorithms. This book examines efficient ways to realize query and update operations on sets of numbers, intervals, or strings by various data structures, including search trees, structures for sets of intervals or piecewise constant functions, orthogonal range search structures, heaps, union-find structures, dynamization and persistence of structures, structures for strings, and hash tables. Instead of relegating data structures to trivial material used to illustrate object-oriented programming methodology, this is the first volume to show data structures as a crucial algorithmic topic. Numerous code examples in C and more than 500 references make *Advanced Data Structures* an indispensable text.

PETER BRASS received a Ph.D. in mathematics at the Technical University of Braunschweig, Germany. He is an associate professor at the City College of New York in the Department of Computer Science and a former Heisenberg Research Fellow of the Free University of Berlin.

Advanced Data Structures

PETER BRASS
City College of New York

CAMBRIDGE
UNIVERSITY PRESS

CAMBRIDGE UNIVERSITY PRESS
Cambridge, New York, Melbourne, Madrid, Cape Town, Singapore, São Paulo, Delhi

Cambridge University Press
32 Avenue of the Americas, New York, NY 10013-2473, USA

www.cambridge.org
Information on this title: www.cambridge.org/9780521880374

First published 2008

Printed in the United States of America.

A catalog record for this publication is available from the British Library.

Library of Congress Cataloging in Publication Data

Brass, Peter.
Advanced data structures / Peter Brass.
p. cm.
Includes bibliographical references and index.
ISBN 978-0-521-88037-4 (hardback)
1. Computer algorithms. I. Title.
QA76.9.A43B73 2008
005.1–dc22 2008021408

ISBN 978-0-521-88037-4 hardback

Dedicated to my parents,
Gisela and Helmut Brass

Contents

Preface

This book is a graduate-level textbook on data structures. A data structure is a method[1] to realize a set of operations on some data. The classical example is to keep track of a set of items, the items identified by key values, so that we can insert and delete (key, item) pairs into the set and find the item with a given key value. A structure supporting these operations is called a dictionary. Dictionaries can be realized in many different ways, with different complexity bounds and various additional operations supported, and indeed many kinds of dictionaries have been proposed and analyzed in literature, and some will be studied in this book.

In general, a data structure is a kind of higher-level instruction in a virtual machine: when an algorithm needs to execute some operations many times, it is reasonable to identify what exactly the needed operations are and how they can be realized in the most efficient way. This is the basic question of data structures: given a set of operations whose intended behavior is known, how should we realize that behavior?

There is no lack of books carrying the words "data structures" in the title, but they merely scratch the surface of the topic, providing only the trivial structures stack and queue, and then some balanced search tree with a large amount of handwaving. Data structures started receiving serious interest in the 1970s, and, in the first half of the 1980s, almost every issue of the *Communications of the ACM* contained a data structure paper. They were considered a central topic, received their own classification in the *Computing Subject Classification*,[2]

[1] This is not a book on object-oriented programming. I use the words "method" and "object" in their normal sense.

[2] Classification code: E.1 data structures. Unfortunately, the *Computing Subject Classification* is too rough to be useful.

and became a standard part of computer science curricula.[3] Wirth titled a book *Data Structures + Algorithms = Programs*, and *Algorithms and Data Structures* became a generic textbook title. But the only monograph on an algorithmic aspect of data structures is the book by Overmars (1983) (which is still in print, a kind of record for an LNCS series book). Data structures received attention in a number of application areas, foremost as index structures in databases. In this context, structures for geometric data have been studied in the monographs of Samet (1990, 2006); the same structures were studied in the computer graphics context in Langetepe and Zachmann (2006). Recently, motivated by bioinformatics applications, string data structures have been much studied. There is a steady stream of publications on data structure theory as part of computational geometry or combinatorial optimization. But in the numerous textbooks, data structures are only viewed as an example application of object-oriented programming, excluding the algorithmic questions of how to really do something nontrivial, with bounds on the worst-case complexity. It is the aim of this book to bring the focus back to data structures as a fundamental subtopic of algorithms. The recently published *Handbook of Data Structures* (Mehta and Sahni 2005) is a step in the same direction.

This book contains real code for many of the data structures we discuss and enough information to implement most of the data structures where we do not provide an implementation. Many textbooks avoid the details, which is one reason that the structures are not used in the places where they should be used. The selection of structures treated in this book is therefore restricted almost everywhere to such structures that work in the pointer-machine model, with the exception of hash tables, which are included for their practical importance. The code is intended as illustration, not as ready-to-use plug-in code; there is certainly no guarantee of correctness. Most of it is available with a minimal testing environment on my homepage.

This book started out as notes for a course I gave in the 2000 winter semester at the Free University Berlin; I thank Christian Knauer, who was my assistant for that course: we both learned a lot. I offered this course again in the fall semesters of 2004–7 as a graduate course at the City College of New York and used it as a base for a summer school on data structures at the Korean Advanced Institute of Science and Technology in July 2006. I finished this book in November 2007.

[3] ABET still lists them as one of five core topics: algorithms, data structures, software design, programming languages, and computer architecture.

I thank Emily Voytek and Günter Rote for finding errors in my code examples, Otfried Cheong for organizing the summer school at KAIST, and the summer school's participants for finding further errors. I thank Christian Knauer and Helmut Brass for literature from excellent mathematical libraries at the Free University Berlin and Technical University Braunschweig, and János Pach for access to the online journals subscribed by the Courant Institute. A project like this book would not have been possible without access to good libraries, and I tried to cite only those papers that I have seen.

This book project has not been supported by any grant-giving agency.

Basic Concepts

A data structure models some abstract object. It implements a number of operations on this object, which usually can be classified into

– creation and deletion operations,
– update operations, and
– query operations.

In the case of the dictionary, we want to create or delete the set itself, update the set by inserting or deleting elements, and query for the existence of an element in the set.

Once it has been created, the object is changed by the update operations. The query operations do not change the abstract object, although they might change the representation of the object in the data structure: this is called an adaptive data structure – it adapts to the query to answer future similar queries faster.

Data structures that allow updates and queries are called dynamic data structures. There are also simpler structures that are created just once for some given object and allow queries but no updates; these are called static data structures. Dynamic data structures are preferable because they are more general, but we also need to discuss static structures because they are useful as building blocks for dynamic structures, and, for some of the more complex objects we encounter, no dynamic structure is known.

We want to find data structures that realize a given abstract object and are fast. The size of structures is another quality measure, but it is usually of less importance. To express speed, we need a measure of comparison; this is the size of the underlying object, not our representation of that object. Notice that a long sequence of update operations can still result in a small object. Our

usual complexity measure is the worst-case complexity; so an operation in a specific data structure has a complexity $O(f(n))$ if, for any state of the data structure reached by a sequence of update operations that produced an object of size n, this operation takes at most time $Cf(n)$ for some C. An alternative but weaker measure is the amortized complexity; an update operation has amortized complexity $O(f(n))$ if there is some function $g(n)$ such that any sequence of m of these operations, during which the size of the underlying object is never larger than n, takes at most time $g(n) + mCf(n)$, so in the average over a long sequence of operations the complexity is bounded by $Cf(n)$.

Some structures are randomized, so the data structure makes some random choices, and the same object and sequence of operations do not always lead to the same steps of the data structure. In that case we analyze the expected complexity of an operation. This expectation is over the random choices of the data structure; the complexity is still the worst case of that expectation over all objects of that size and possible operations.

In some situations, we cannot expect a nontrivial complexity bound of type $O(f(n))$ because the operation might give a large answer. The size of the answer is the output complexity of the operation, and, for operations that sometimes have a large output complexity, we are interested in output-sensitive methods, which are fast when the output is small. An operation has output-sensitive complexity $O(f(n) + k)$ if, on an object of size n that requires an output of size k, the operation takes at most time $C(f(n) + k)$.

For dynamic data structures, the time to create the structure for an empty object is usually constant, so we are mainly interested in the update and query times. The time to delete a structure of size n is almost always $O(n)$. For static data structures we already create an object of size n, so there we are interested in the creation time, known as preprocessing time, and the query time.

In this book, $\log_a n$ denotes the logarithm to base a; if no base is specified, we use base 2.

We use the Bourbaki-style notation for closed, half-open, and open intervals, where $[a, b]$ is the closed interval from a to b, $]a, b[$ is the open interval, and the half-open intervals are $]a, b]$, missing the first point, and $[a, b[$, missing the last point.

Similar to the $O(\cdot)$-notation for upper bounds mentioned earlier, we also use the $\Omega(\cdot)$ for lower bounds and $\Theta(\cdot)$ for simultaneous upper and lower bounds. A nonnegative function f is $O(g(n))$, or $\Omega(g(n))$, if for some positive C and all sufficiently large n holds $f(n) \leq Cg(n)$, or $f(n) \geq Cg(n)$, respectively. And f is $\Theta(g(n))$ if it is simultaneously $O(g(n))$ and $\Omega(g(n))$. Here "sufficiently large" means that $g(n)$ needs to be defined and positive.

Code Examples

The code examples in this book are given in standard C. For the readers used to some other imperative programming language, most constructs are self-explanatory.

In the code examples, `=` denotes the assignment and `==` the equality test. Outside the code examples, we will continue to use = in the normal way.

The Boolean operators for "not," "and," "or" are `!`, `&&`, `||`, respectively, and `%` denotes the modulo operator.

Pointers are dereferenced with `*`, so if `pt` is a pointer to a memory location (usually a variable), then `*pt` is that memory location. Pointers have a type to determine how the content of that memory location should be interpreted. To declare a pointer, one declares the type of the memory location it points to, so "`int *pt;`" declares `pt` to be a pointer to an `int`. Pointers are themselves variables; they can be assigned, and it is also possible to add integers to a pointer (pointer arithmetic). If `pt` points to a memory object of a certain type, then `pt+1` points to the next memory location for an object of that type; this is equivalent to treating the memory as a big array of objects of that type. `NULL` is a pointer that does not point to any valid memory object, so it can be used as a special mark in comparisons.

Structures are user-defined data types that have several components. The components themselves have a type and a name, and they can be of any type, including other structures. The structure cannot have itself as a type of a component, because that would generate an unbounded recursion. But it can have a pointer to an object of its own type as component; indeed, such structures are the main tool of data structure theory. A variable whose type is a structure can be assigned and used like any other variable. If `z` is a variable of type `C`, and we define this type by

```
typedef struct { float x; float y; } C,
```

then the components of `z` are `z.x` and `z.y`, which are two variables of type `float`. If `zpt` is declared as pointer to an object of type `C` (by `C *zpt;`), then the components of the object that `zpt` points to are `(*zpt).x` and `(*zpt).y`. Because this is a frequently used combination, dereferencing a pointer and selecting a component, there is an alternative notation `zpt->x` and `zpt->y`. This is equivalent, but preferable, because it avoids the operator priority problem: dereferencing has lower priority than component selection, so `(*zpt).x` is not the same as `*zpt.x`.

We avoid writing the functions recursively, although in some cases this might simplify the code. But the overhead of a recursive function call is significant

and thus conflicts with the general aim of highest efficiency in data structures. We do not practice any similar restrictions for nonrecursive functions; a good compiler will expand them as inline functions, avoiding the function call, or they could be written as macro functions.

In the text we will also frequently use the name of a pointer for the object to which it points.