1 Introduction

1.1 The Course of This Course

In a nutshell, this course has two foci: The first is the efficient management and manipulation of data. The second is the ability to implement methods for efficient management and manipulation of data in a modern high-level programming language, which in this case happens to be Java™. This is not intended to be "a course in Java"; rather, it is a course in the use of data structures and algorithms that uses Java as the vehicle for turning concepts into the details of implementations. In the final analysis, most people do not get paid to talk about how one might compute something; they get paid to facilitate the actual computation of something. Really knowing the details of implementation is thus very important. Students in computer science should actually be happy that computers, compilers, and programming languages are as rigid as they are. This allows the rigorous testing of ideas, because "the computer" is the final arbiter of correctness, and one can do one's own testing for correctness, unlike, say, mathematics, chemistry, or the law.[1]

Many of the basic problems of data structures immediately present themselves when dealing even with so simple a thing as an address book. Much of the purpose of this text and the course it is intended to support is to present algorithms and methods that are better in one way or another than the naive approaches that one might take without thinking very hard.

We assume that a student who has entered an intermediate course already has experience at an introductory level in the manipulation of data and in writing programs to manipulate data. The most fundamental, and most naive, view of data is usually that of a *flat file*. A flat file is simply a two-dimensional data array like a spreadsheet. Each line can be viewed as a *record*. Each column can be viewed as a *field*. In a simple example, one can think of an address book. Each record is an individual "person," and each record possesses fields for last name, first name, street address, city, state, zip code, telephone number, and so forth. Actions that

[1] Actually, there is nothing in the law that says one cannot determine what is legal or illegal by the experimental method, but the penalties can be more substantial than having the compiler simply inform you that your program has a bug.

© Jones & Bartlett Learning, LLC. NOT FOR SALE OR DISTRIBUTION.

would reasonably be taken on such a structure would be to sort it into alphabetical order, to add new records and delete old ones, and to retrieve entries based on one or more *search keys* like last name or city of residence. In Chapter 3, we will present the basics of a naive flat file implementation, and then in later chapters we will introduce the structures and algorithms that allow for more sophisticated and more efficient management of the data.

■ 1.1.1 Layers of Software

A central theme in the development of software in a modern setting is that software exists in layers. This is critical to understanding concepts of how and why software is developed the way it is and why there are right ways and wrong ways to develop code.

If you, for example, are the programmer developing an address book implementation, your job is to develop code that handles "address-book-ness." This will include adding, deleting, and editing records; sorting, displaying, and printing copies; perhaps adding and deleting fields in the record; and (in many current applications) linking to web pages or sending text or making phone calls. What you as the address book programmer ought not to be concerned about is how the data is stored in detail. That takes place one level down in the software hierarchy. Although you need to be aware of the *existence* of the data storage methods that have been chosen for use, your code need not be concerned with the *implementation* of those storage methods. Implementation of the storage methods should take place one level down, and that code should be written so as to be independent of the application that is using it.

This layering of code allows programmers to concentrate on the tasks at hand without being distracted by other issues, and it allows code to be used by different programs. You have by now used library code for things like input and output. Your program need not be concerned with *how* the `println` statement gets your data to the console; it need only be confident that the code will accomplish that task correctly and consistently. In this course, we will extend that concept to higher structures that you write as well as to further structures built into Java in the large.

■ 1.1.2 Algorithms and Efficiency

Computers demonstrate their real value not when doing small amounts of work on small amounts of data, but when doing large amounts of work on large amounts of data or when doing the same small task over and over again. Modern computers are very fast, much faster than their predecessors of even 10 or 20 years ago. But as fast as computers are, it is still usually the case that we *do* need to make programs run at least reasonably efficiently. For this reason, we will cover some of the background of how we measure the efficiency of an algorithm or program and why some algorithms

© Jones & Bartlett Learning, LLC. NOT FOR SALE OR DISTRIBUTION.

are inherently more efficient than others. Almost invariably, no matter how powerful the computer, a person intending serious use of the computer to support some activity will run up against the performance limitations of the machine and will need to use better algorithms.

There are two ways in which computational performance can be improved: One can change from an inefficient algorithm to an efficient algorithm, or one can change from a poor implementation to a good implementation. It is usually argued that the real improvements in performance (factors of 10 and 100) come from improved algorithms, and that improved implementations usually provide factors of perhaps 2 to 4. One should therefore look for a good algorithm and implement that, and one should do a good implementation. That being said, it's also possible to overdo both of these. The really good algorithms will also be more complicated to implement (and thus more prone to error), and the benefit of the algorithms is usually only on large input sets. If the input data is small, then a good algorithm is better than a naive algorithm and a great algorithm is probably overkill. Similarly, if a small improvement in performance can be achieved by an implementation that makes the code impossible to read, understand, and maintain, then maybe that's not good either.

1.1.3 Sorting

It will normally be the case that we want to keep an address book in sorted order, usually by last name, because that is how we identify records for retrieval. It has been estimated that perhaps a third of the CPU cycles expended in all of computing are spent in sorting records to keep them accessible in a desired order. For this reason, sorting records is an extremely important subject in data structures and in computing in general, and thus we will spend an entire chapter on sorting.

There are sorting methods that are very simple to implement. Generally, since there is no free lunch in this world, these are also the methods that are not very good. The good sorting methods are somewhat more complicated, but it turns out that a reasonably simple method—quicksort—is actually quite effective. Although we can prove (and will prove) that *quicksort* is not the best method for all sorting problems, it is quite good on average. We will spend a little more time on the bigger picture of sorting because the comparison of the major sorting methods—quicksort, *heapsort*, and *mergesort*—provides a good introduction into the trade-offs that good software people need to make

- between good algorithms that might be hard to implement and poorer algorithms that are easy to implement;
- between the *best-case behavior*, the *worst-case behavior*, and the *average-case behavior* of algorithms;

© Jones & Bartlett Learning, LLC. NOT FOR SALE OR DISTRIBUTION.

- and between the different implementation characteristics of algorithms when we move beyond a "theoretical" description of the algorithm and actually write a program that runs on a real computer.

1.1.4 Maintaining Sorted Order

Change is a constant fact of life. No matter how careful one is to anticipate all possibilities, something unanticipated will also happen. In the case of data, we know that change will happen. As soon as we create an address book with all our contacts in it, we will encounter a new individual who needs to be added to the file. Since we will want to maintain our contacts in sorted order, we will probably need to insert the new contact into the data file in the correct location.

If our flat file is in fact a spreadsheet, we could do this by adding the new record at the bottom and then using the "sort" function to move all the data into the proper locations. In a simple spreadsheet, the sort will quite literally move all the records, and for an individual's address book of a few hundred records, this can be acceptable behavior. However, if the data records are the entire database of the U.S. Internal Revenue Service, then physically moving all the data on disk so as to insert one record is not acceptable. Part of the study of data structures, therefore, is the study of how to maintain sorted order in the presence of change.

1.1.5 Indexing, Search, and Retrieval

Although in many instances there is a natural *primary key* on which to retrieve records (such as last name for our address book), it is often important to be able to retrieve records from a (conceptual) flat file in an order other than the order in which they are stored conceptually. If one is maintaining a mailing list, then in addition to having records sorted by last name, one may well need to be able to retrieve records sorted by zip code so as to get the benefit of bulk mail postage rates. To do this, one would *invert* the file on the zip code field and create an index that can be used for retrieval on the secondary key instead of the last-name primary key.

More importantly, one of the primary features of any complex computer program is that it will search for and retrieve a data item, based on a *key*, from a "collection" of data. The key might be a Social Security number, a name, or even a request for "the next" item in some sorted order. We will therefore deal with the organization of data so that search and retrieval of data can be done efficiently.

In working with data, what matters a great deal is whether or not that data is structured. A retail receipt for purchases at the grocery store represents structured data. Each such record has date, time, and store information, as well as some number of lines of transaction information for the item purchased, the individual price, and the number of items that were purchased. One can easily imagine a

© Jones & Bartlett Learning, LLC. NOT FOR SALE OR DISTRIBUTION.

template for a receipt record. The boilerplate store information would be easy to add; the date and time would change but would be easy to obtain from an internal clock; and the only complexity comes from the fact that one has a variable number of items to record.

This kind of structured data is very different from a web page or a collection of web pages forming an entire website, for example, which would have free text, links, internal horizontal references, etc. In a retail store, one would assume that the list of items for sale could be found somewhere in a file on a master computer. For a randomly chosen website, though, no master list of the words used on the site would be expected to exist. That list of words would have to be built dynamically by reading through the pages. With a retail transaction, we expect header and footer information and a list of items and prices in the middle. With the pages of a website, we have no reason to know what to expect in terms of the internal and external URL references. We have no reason to know the length of the pages or where the content will occur. If we are to build a data structure to hold that information, we must use a structure that can adapt automatically as we read the data.

1.1.6 Dynamic Versus Static Behavior

As soon as one introduces the notion of change to a data file (or more generally to a data object) and sets about to decide upon a data structure for that file, one is forced to consider the operational characteristics of the dynamic activity on that data. The questions that need to be addressed include the following:

- What is the size of the data object?
- What is the size of the object relative to the ongoing change to the object?
- What happens to data items when they are “deleted” from the collection of data?

With the data file of IRS records on U.S. taxpayers, for example, it is probably a reasonable assumption that the base file is enormous relative to the daily changes to the file. There are hundreds of millions of taxpayer IDs, and no doubt billions of individual records of tax payments, but the rate of change of the data object is probably relatively small. If the data object consists of the active processes in a computer, however, the number of processes will be small (in the low hundreds at most), but a large number of the processes will be very short lived and the turnover will be very high. Similarly, the servers of an ISP will have at any time a collection of individual packets of Internet transmissions, but the set of packets at any given time will be constantly changing. Different rates of change will require different data structures.

© Jones & Bartlett Learning, LLC. NOT FOR SALE OR DISTRIBUTION.

What are the access and retrieval patterns? Is the data[2] stored once and retrieved a large number of times, as in an address book? Or is the data stored on an ongoing basis and then retrieved only occasionally? Is there a need to make retrieval of some (more popular) records more efficient than the retrieval of other records? Is it the case that there are usually only a few updates to be made, as with a university transcript database, but then there are times (such as the end of the semester) when bulk updates occur?

Google™, for example, will want to index everything, but part of the success of Google has been its ability to quickly point to what the user community has determined to be "relevant" data. The use of last name as the primary key in a personal address book is due to the fact that this is (for most people, at least) the normal way in which the data would be identified and searched for. Retrieval by zip code for bulk mailing purposes can be useful, but it is not going to be the common method for retrieval of names and addresses.

Similarly, in an operating system, the available processes to execute can probably be sorted by priority, but with only one or perhaps two CPUs, there will be a high priority placed on being able to identify very quickly "the next" process in the data structure to execute since any time spent on deciding which process to execute will not be viewed as productive work for the user. A similar situation will exist if the data object that is being built is a game tree of potential moves. We will want to build as large a tree as possible of our potential moves, the opponent's potential responses, our responses to the responses, and so forth, together with a measure of our game status after each move and countermove.

■ 1.1.7 Memory and Bandwidth

Increasingly, software is focusing on mobile devices. Because these devices are mobile, and usually mass-marketed, they have limited resources in processor power and in memory or longer-term storage. As mobile devices, they gain their utility largely by being connected wirelessly, which means that the bandwidth from the device to the rest of the universe is limited. Anyone who has downloaded large files or accessed the Internet from a slow telephone line knows that some things just don't work well due to a mismatch of the program and the bandwidth of the machine. This is most apparent with images and video. It is virtually impossible to access mere text in sufficient quantity to make bandwidth an issue, but it is relatively easy to do this

[2]The author generally tries to be strict in his grammar, and he is aware that "data" is officially the plural of "datum," is therefore plural, and should take the plural verb "are" and not the singular verb "is." However, even though the author generally tries to be strict about correct grammar, this is one instance in which the common singular usage, treating data as a mass noun, especially with regard to computer data, seems appropriate.

© Jones & Bartlett Learning, LLC. NOT FOR SALE OR DISTRIBUTION.

with images. Anyone who has seen a video buffered up or played back in a bursty fashion will recognize the problem.

To counteract the problems with bandwidth, most programs for mobile devices must make careful use of memory and data structures, freeing up and then reusing memory space when possible, and delaying execution of certain functions for as long as possible.

Consider something as simple as a slideshow of photos for a museum tour. If the program is written to run on mobile devices owned by the museum, the photos can be downloaded to the devices long in advance and kept in a memory card. If, however, a museum visitor can use her own smart mobile device, then the program designer has to consider whether it will be acceptable to users to have to wait 2 or 3 minutes to download the entire collection of images and then display them. Or should the download be delayed as long as possible, in hopes that the visitor will linger over photo n long enough to permit photo $n+1$ to be downloaded just before it is needed?

■ 1.1.8 Heuristics

Finally, we recognize that there are times when the best becomes the enemy of the good. In any given application, how good is good enough? We have already mentioned that quicksort, to be covered later, is usually the "standard" sorting method used for practical applications, not because it is theoretically the best method available (it is not) but because it is a relatively simple method with good average-case behavior.

Similarly, it is probably unusual for someone to print a new paper copy of an address list just because one person has been added to the electronic list. More likely the electronic list is updated and the paper copy is simply annotated with a handwritten reference. When so many changes to the paper copy have been made that the paper copy is too messy to be convenient, a new paper copy will be printed. In truly serious data-intensive applications, the same principle might well apply. If there is enormous effort involved, for example, in a one-time sort of the entire data object, then one might plan to maintain the changes as an addendum to the main object and then incorporate the changes in a block only at intervals, perhaps daily, weekly, or monthly. For such applications, the heuristics of how to do *incremental* updates can be very important.

A *heuristic* in computer science is a general guideline, based on experience and an understanding of the problem, that leads to the implementation of specific algorithms or design features in the software meant to solve the problem or provide the service. Especially when performance is an issue, we usually write programs whose execution is based on normal, average use. We will include the code to deal with *pathological cases* (because we have to cover all possible inputs), but we will

© Jones & Bartlett Learning, LLC. NOT FOR SALE OR DISTRIBUTION.

not usually write code that assumes that the one-in-a-billion worst case is going to happen. For example, the first time we create a sorted telephone book, we might want to assume that the records are in a reasonably random order, but if we are adding new records to a telephone-book-sized address list, we can safely assume that we are starting with a large list that is already sorted and then adding a small number of records to that list. These assumptions might well lead to the use of a different sorting algorithm for the incremental updates than was used for creating the list in the first place.

A different version of the heuristics issue is that programming for performance requires identifying the most serious bottlenecks and then applying some better brainpower to those bottlenecks first. There is probably no point in using a complicated algorithm for a task that is done infrequently and is not time-critical. In software, as in life, there is always a list of the most important things to be accomplished, and we should be attacking those before we go about making cosmetic changes or changes that will have no real effect.

1.1.9 Templates, Generics, and Abstract Classes

We have described a flat file that comprises records (lines in a spreadsheet), each record of which has fields (the columns). In any real programming situation, field items will exist as strings, integers, floating-point numbers, and other less primitive types. To avoid writing almost the same code over and over again, Java can provide *generic* constructs and `abstract` classes that will simplify the coding process (that is, the process will be simpler once one learns how to use generics and abstract classes). In sorting, for example, the structure of a sort is independent of whether one is sorting integers as integers or sorting strings (like last names) lexicographically; the base step in sorting an array into "increasing" order is to compare two data items and exchange the order of the two items if necessary so that the "smaller" item comes first. Good programming practice will therefore put off until the very last moment any code that specifically deals with one data type or another, and only at the very last moment use an appropriate compare-and-exchange method for two data items.

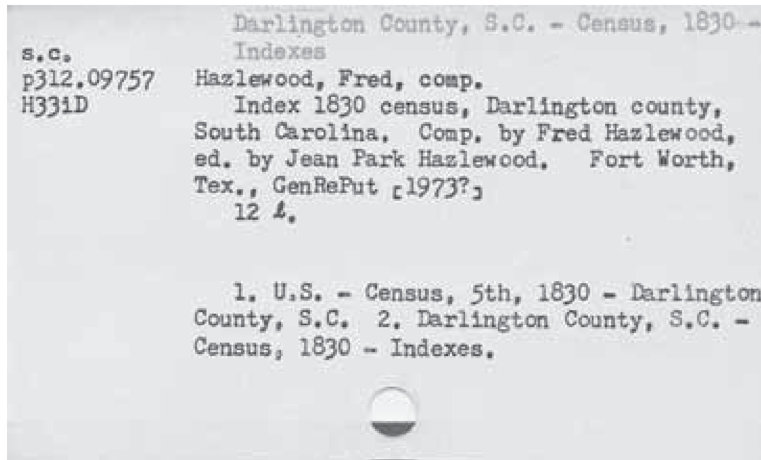
One issue of data structures that does not necessarily present itself in a flat file but is so important that it warrants mention here is the nature of tree structures and then eventually of recursion. One canonical use of a tree structure is the directory structure on a modern computer. In Unix, the "root" data structure is the "/" directory, and all the directories and files live underneath that. Users will have a "home" directory and create under that various files and subdirectories, with the subdirectories then having files and subdirectories, and so forth. In Windows, most user data starts at the My Documents folder (Windows uses the term "folder" instead of "directory") and descends from there.

© Jones & Bartlett Learning, LLC. NOT FOR SALE OR DISTRIBUTION.

1.2 Examples

1.2.1 Card Catalogs

Although it is not necessarily one of the more exciting applications of data structures, the card catalog of a library is one example of a large data file that needs to be properly organized to be useful to the user community. These used to be physical cards, as shown in Figure 1.1, but are now almost always kept entirely online in large libraries.



Darlington County, S.C. - Census, 1830 - Indexes

s.c.
p312.09757
H331D

Hazlewood, Fred, comp.
Index 1830 census, Darlington county, South Carolina. Comp. by Fred Hazlewood, ed. by Jean Park Hazlewood. Fort Worth, Tex., GenRePut [1973?]
12 ℓ.

1. U.S. - Census, 5th, 1830 - Darlington County, S.C. 2. Darlington County, S.C. - Census, 1830 - Indexes.

FIGURE 1.1 ■ A card from a library card catalog.

The first thing to note is that the card catalog data is stored essentially only once (unless there are corrections to be made, and these should be rare). Except for a data field indicating whether the document is or is not checked out of the library (and the possible checkout history that we will ignore), the data file, although perhaps large (libraries in major universities have collections numbering in the millions of volumes), is very static. The issue with this kind of data is not constant updating but rather that the data can be accessed by multiple search keys. In this case, access by author name, document title, Library of Congress classification number, keywords, perhaps accession date, and perhaps also due date for documents checked out, would be necessary. We will want to use a data structure that permits records to be added over time, and there will be records that occasionally get deleted, but the major issue is not the change in the data file but in the efficient retrieval of record information.

Data such as card catalog data is usually very highly structured. Indeed, substantial effort has been expended by the U.S. Library of Congress to standardize card catalog data so that records are uniform in all libraries in the country.

© Jones & Bartlett Learning, LLC. NOT FOR SALE OR DISTRIBUTION.

1.2.2 Student Records

A classic kind of record to be stored and manipulated are the student records at educational institutions. Each record contains several different kinds of data. Name, student ID number, and such are going to be almost static. Address information, major, and such will need to be updatable, but this together with the identity information will be data "local to the student." This will differ from the transcript information. Both for privacy and efficiency reasons some information is likely to be stored in different files from the address information. We would also not expect the individual course grades to be stored with the student information; rather, we would expect all the grade information for a given course to be stored in one place, with the student record indexing into the course data.

In the case of student records, then, we have the likelihood that, unlike the card catalog information for a document in a library, "the record" for a given student will not consist of a single record, in a single array, in a single Java class (assuming Java to be the implementation language). Indeed, in most significant implementations, a single "record" is likely to be the aggregation of fields from several different stored collections. One reason for this is to prevent duplication of data because duplication leads to the problem of consistency across instances. If one list of grades by class were kept and a separate list of grades by student existed, then changes would have to occur in more than one location. This is always something to be avoided in designing software.

1.2.3 Retail Transactions

A number of standard data processing environments constitute *transaction processing*. For example, both a large retailer, such as Walmart, and a credit card processor, such as Visa, deal with millions of individual transaction records every day that come in on a pretty much nonstop basis. Characteristics of the data records will be that they are small, with several fixed fields (date, payment method, purchase location, and such), and a variable number of individual item purchases made. The data records need to cataloged and stored, with multiple indexing and retrieval patterns made possible, but it is unlikely that the individual items will be accessed randomly or very often.

1.2.4 Packet Traffic

Internet packets resemble transaction records in many ways. One characteristic of Internet packets is that they are more or less fixed size (a primary reason for packetizing an Internet transmission is to keep the packet size small) and have a number of fixed fields (source, destination, etc.). One characteristic somewhat

© Jones & Bartlett Learning, LLC. NOT FOR SALE OR DISTRIBUTION.

unique to packet traffic is that what the user views as a single transmission is broken into packets at the source to be sent, possibly through different routes, to the destination. The packets arrive at the destination in no particular order and then must be reassembled in the proper order before the complete message is delivered to the destination user.

This kind of data is different from all those mentioned previously in that the data records (packets) are not standalone items but must be aggregated into complete transmissions in real time at high speed. Student records are expected to be persistent and long term, but Internet messages are packetized, sent, collected, and then discarded. This kind of processing requires a much different data structure from that used previously in order to accommodate these different needs.

1.2.5 Process Queues

Similar in some ways to Internet packet traffic are the process queues managed by the operating system in any modern computer. In even a standard desktop there will be dozens of processes in a *state of execution* at any point in time. Some of these are processes that are always running; some are processes created when the user fires up a word processor, an edit window, an Internet browser, and so forth. The task of the scheduler for the operating system is to determine which process to execute next. As with packet traffic, these process queues must be managed in real time with as little effort as possible expended by the scheduler. Processes need to have their priorities updated at intervals, and the effects of the updated priorities need to be available immediately. The data structure used to implement the process queues needs to be highly dynamic and easily changed, and the processes with the highest priorities need to be quickly accessible.

1.2.6 Google

All the preceding examples of data are of *fixed-format records*. These are records that can be thought of in spreadsheet form; each record has a list of fields, and each field has a specific data type. Although some fields might be of variable length, the sequence of the fields and the data type for each field are the same across all records. Further, there is some reason to believe that the data as stored will actually make sense because the records have been created by an authoritative source or by means of software that has presumably been written so as to produce reasonably correct records.

All that goes out the window when one considers the Google problem of analyzing all the web pages in the world. Web pages are not fixed-format; they are highly changeable over time; they are not created by an authoritative source; they may be rife with misspellings or typographical errors; and they are not guaranteed to

© Jones & Bartlett Learning, LLC. NOT FOR SALE OR DISTRIBUTION.

be organized. All these complications require greater thought when deciding how to craft a data structure that contains the information that must be processed and presented to a user.

1.2.7 Game Trees

A large number of computing applications can be described simply as involving the optimizing (either minimizing or maximizing) of an *objective function* over a *tree search*. A game tree for chess or a similar game falls into this category. At any point in the chess game, White has a number of possible moves. Black has a number of possible responses, to which White has responses, and so forth. White's objective function for the game would be a function that would assign to any given board position a "goodness" value for White for that position; this goodness function is the objective function, and White's game-playing strategy is to determine which moves maximize that function. Given any possible move, one can recompute the goodness of the new board position, then compute the goodness of the board position after Black's response to the particular move, and so on. Structurally, the options resemble the branching of a tree from the *root* (the current position) down through the options of move and countermove.

Such a tree search is *combinatorially explosive* in the total number of possibilities, because even if we restrict ourselves to a fixed number k of possible moves at every level, the number of possible moves for n levels is k^n and thus grows exponentially with n. Most game programs are thus implemented with some sort of *heuristic search* in which the heuristics of the search strategy limit the number of choices at each decision point and thus limit the total number of possible paths that might be explored.

1.2.8 Phylogenetics

Phylogenetics is the subdiscipline of biology in which one computes backward through time from a list of current species (called taxa) to determine the best possible evolutionary tree back to the common ancestors of today's taxa. This is a variant of a classic example of tree search; the current taxa are the leaves of the tree and, moving backward through time to a common ancestor, the length of each edge is proportional to time. The problem is that, as with most tree searches, the number of possible paths is astronomically huge and thus efficient data structures as well as heuristic algorithms must be applied in order to perform any sort of serious computation.

© Jones & Bartlett Learning, LLC. NOT FOR SALE OR DISTRIBUTION.

1.3 Summary

As we shall see in Chapter 3, a standard way to begin to program an application is to use the simple and naive algorithms and data structures. These would be simple to program, would work correctly, and would almost certainly work so inefficiently as to be unacceptable in practice.

An implementation under development will usually move from simple-but-inefficient to efficient-but-harder-to-program in stages, with each stage providing better efficiency at a cost in complexity. This complexity usually involves management of subscripts, control over memory usage, and great care with endpoint conditions and null values. To control this complexity, most modern programming languages come bundled with solid implementations of the standard data structures.

In Java, these are the classes in the Java Collections Framework. Virtually all the data structures in this text exist in the JCF. If mere use of the structures and algorithms were all that was needed, there would be no need for you as the programmer to reinvent the code for a linked list, for example. That code already exists in the JCF, and it is highly unlikely that your code will be as general purpose, as robust, and as thoroughly tested as that in the JCF.

You, the programmer, do not really need to rewrite the code; in a professional setting, you would probably choose to use the existing code, leaving you free to concentrate on implementing code for the actual application instead of simply for the underlying data structures and algorithms. However, really understanding the structures and algorithms requires a deeper knowledge than that of mere usage, and you are unlikely to really learn how these data structures work without implementing them in code yourself. We have thus written the text and set the exercises as if the reader were going to implement all the data structures and algorithms. Many of them are quite similar, and perhaps it is not necessary to do all of them; having implemented the code for a stack, perhaps implementing the code for a queue is not as good a use of your time as would using the JCF queue in a more interesting application. Nonetheless, we encourage practice in writing code. We believe strongly that gaining a real understanding both of the programming process and of the algorithms and data structures necessary for applications of significant size requires the trial and error of programming and the knowledge acquired through the fingertips in the school of hard knocks.

© Jones & Bartlett Learning, LLC. NOT FOR SALE OR DISTRIBUTION.

© Jones & Bartlett Learning, LLC. NOT FOR SALE OR DISTRIBUTION.