Engineering Algorithms and Data Structures

Ken Clowes

2002

Version 1.1 Jan 24, 2003

Contents

Preface

This book is used as the course notes for the Ryerson Electrical and Computer Engineering second year course *ELE 428*—Engineering Algorithms and Data Structures.

The book describes classic algorithms and data structures including stacks, queues, priority queues, trees, graphs and hash tables. The emphasis is on understanding the various algorithms in language-independent way and to be able to analyze their performance.

In addition to basic understanding, however, we also place importance on the ability to design and implement algorithms in a programming language in a competent, professional manner. The primary programming language we use is C.

The book is divided into three parts: Part I—*Fundamentals*—gives an overview of the definition of algorithms and data structures, how they are presented in this book, and how to analyze their performance. Part II—*Data Structures*—forms the core of the book. The basic data structures and associated algorithms for searching, sorting, and modifying sets, bags and graphs are described and implemented. Part III—*Projects*—puts the ideas presented in the first two parts to practical use in engineering applications.

Details

> *Just the facts, ma'am*
>
> —Sgt. Friday on Dragnet

We now give a brief overview of the book's structure and suggestions on how to read it.

Part I is an overview—many topics may be a review of ideas the reader is already familiar with.

Version 1.1 Jan 24, 2003

Chapter 1 lays the foundations for the development, definition, analysis and implementation of algorithms. Simple sort algorithms (selection sort and merge sort) are used as the primary examples. (Other algorithms including insertion sort and quick sort are treated in the problems.) The implementations in C used in this chapter illustrate the general coding and testing standards we use throughout the book. Most of this chapter should be a review for readers.

Chapter 2 is a review of recursive programming techniques.

Chapter 3 puts recursion to important practical use by developing basic translators and interpreters for simple formal languages defined by the BNF meta-language.

Chapter 4 defines various forms of asymptotic notation useful in comparing the performance of different algorithms. Various examples are given as well as some basic techniques for solving recurrences.

Chapter 5 introduces the concept of Abstract Data Type and how it can be implemented in C. This chapter is mainly concerned with programming techniques. We also take the idea of ADT and combine it with some concepts from Object-Oriented design and programming. These techniques are used extensively in the projects.

Part II forms the core of the book. The topics include:

Chapter 6 introduces fundamental linear data structures including stacks, queues and priority queues including several examples of how these structures are used and how they can be implemented.

Chapter 7 introduces the most important non-linear data structure in computer programming—*trees*.

Chapter 8 looks at balanced binary search trees (BSTs) including Red-Black and AVL trees.

Chapter 9 introduces hash tables, often the best way to organize data for rapid searching. We also relate these ideas to hardware organizations such as cache memory and look-aside buffers used in virtual memory systems.

Chapter 10 examines basic graph data structures.

Chapter 11 gives a rapid overview of topics in the theory of computation such as Turing Machines, the "Halting Problem", NP-complete problems, etc. This chapter is *not* currently part of the ELE428 syllabus.

Chapter 12 is a short summary of classic approaches to algorithm design. Some topics are: dynamic programming, heuristic methods, etc. This chapter is *not* currently part of the ELE428 syllabus.

Chapter 13 is an overview of techniques for implementing algorithms in

hardware that allows readers to combine their understanding of algorithms and data structures with their skills in digital system design. Some of the topics include: basic sequential implementation and pipelining, cache memory system as a form of hashing, sorting networks, FFT networks, and general strategies.

Part III contains projects that electrical and computer engineering students may find interesting. The projects use many of the basic data structures and algorithms described in the book and apply them to larger problems.

Chapter 14 contains the design and implementation of a general purpose event-driven simulator.

Chapter 15 contains the design and implementation of a package of useful modules for analyzing and synthesizing basic Boolean combinational logic circuits.

Finally, the Appendices contain auxiliary information.

Appendix A explains the coding and organizational standards recommended for students as well as how this book itself was organized.

Appendix B is a review of elementary data structure and pointer use (including memory allocation and manipulation) for the C programming language.

Appendix C is a review of basic organizational strategies for modules making up a software product.

Appendix D gives the answers to most of the problems.

Appendix E gives the complete source code for all the programs discussed in the book.

There is also an annotated Bibliography, an Index and a Colophon (describing the software tools I used in writing the book).

Author's note: This Appendix will **not** be part of the final book. It is included in the draft version for easy reference. (Of course, all the code will always be available electronically.)

Philosophy

Just the FAQs, ma'am

—Sgt. Dayfry on NETdrag

The philosophy of the book can be explained as answers to "Frequently Asked Questions" given below. (These are also questions I frequently ask myself.)

Why use C—a dinosaur language?

C is a simple language that is appropriate for analyzing and implementing basic algorithms. And, far from being a dinosaur, it is the programming

language of choice in one of the most popular products that has hit the market in recent years—the Palm Pilot personal digital assistant. ...Oh, and by the way, have you heard of *Linux*...

Why use English instead of pseudo-code (or Smalltalk or lisp or ...) to describe algorithms?

I use English because it emphasizes the basic structure of an algorithm without reference to any particular programming language and avoids the danger of chasing the latest "trendy" language. A competent programmer in any language should be able to express what they are doing in English and be able to convert a clear description of any algorithm to their favorite programming language.

What does the word "engineering" mean in "Engineering Algorithms and Data Structures"?

As a trivial example, I occasionally use j, rather than i, to represent $\sqrt{-1}$ and I use the terms "nodes" and "arcs" in graph theory rather than the words "vertices" and "edges" usually found in mathematical treatments.

More seriously, I have tried to take examples from engineering (especially digital systems) to illustrate general principles. In particular, the projects are designed with engineering applications and concepts in mind.

What are your favorite books on data structures, algorithms and programming? Why did you write another one?

My favorite books on data structures and algorithms are *Introduction to Algorithms*[THC90] by Cormen, Leiserson, and Rivest and Donald Knuth's three volume opus *The Art of Computer Programming*[Knu97a]. They are more advanced and mathematical than this book (especially Knuth).

In one sense, my book is a simplified version of these classics. I hope, however, that it is not too "dumbed down". Indeed, I hope that I have covered enough basics in this book so that interested readers could approach these more advanced texts on their own.

For ideas on programming philosophy and practice, I like *The Mythical Man-Month*[FP95] by Brooks, *The Practice of Programming*[KP99] by Kernighan and Pike and *Programming Pearls*[Ben99] by Jon Bentley. I also enjoy *The Design of Everyday Things*[Nor90] by Donald Norman for insights on how to design things in general and software in particular that is usable.

For C Programming, my preferred references are *The C Programming Language*[KR88] by Kernighan and Ritchie and *C, A Reference Manual*[HJ91] by Harbison and Steele.

What is your favorite programming language?

Right now (and for the past 3–4 years), it is Java... but things change...

How did you choose the examples for the book?

Most of the examples are either classics or have relevance to other branches of computer/electrical engineering at roughly the same level as is addressed in this book (i.e. second or third year).

Classic examples (such as the *Towers of Hanoi* problem or *ways to count change*) are important not because anyone will ever have to write a program to perform these things, but because there is a wealth of literature about them and most students in a computer/electrical engineering/science program have encountered these classic problems during their studies. This creates a common vocabulary amongst practicing programmers, analysts and engineers that can be exploited to communicate ideas succinctly.

Why did you choose TEX (LATEX) to write the book?

Because I am old(-fashioned). I am used to this environment and find that it produces high-quality typeset hardcopy (especially for math) and it is usable as a front-end to an HTML translator to produce web pages that I have some control over.

Notes on the problems

There is an extensive set of problems and I have written solutions for *all*[1] of them. Most of the solutions can be found in Appendix D. However, some solutions have not been published; they are only available to your professors.

Some of the problems explore concepts in greater depth than is given in the text and the answers are more complete. There is no indication (currently) about which problems are simple drills or exercises and which have longer answers that readers may benefit from studying. (All you can do for now is look at the answers in Appendix D and note which ones seem particularly long.)

There is also a private (not distributed) bank of questions that we can use for quiz and and exam purposes.

[1] This is not entirely true... but I am working on it. I hope to eliminate this footnote before the end of term (April 2000)

Notes on the source code

The complete source code for all examples and problems is available on-line. The source code integrated with the text is derived from working code, but is not always identical. For example, error-detection (checking return values for sanity, etc.) is often removed so that the basic algorithm stands out. Other changes include different or more comments and eliminating the use of the `eprintf` library functions.

I also suggest that readers at least skim the coding standards (Appendix A) before serious examination of the source code.

Typographical conventions

We use `this font` for C source code or other things whose precise meaning is directly expressed as a well-defined sequence of characters (including things you might type or see on a dumb terminal). (For example, most of the times this font is used, the contents would not change if the entire book were translated into another language.)

Another convention used is to...

> isolate details (that may be of interest to only some readers) without breaking the linear flow of the text... Readers who feel comfortable with the ideas expressed previously can simply skip the details described here...

The offset text above is an example. (It is set in smaller type and is indented.) In a sense, it is "like a footnote"[2]. It is really more like a "sidebar" in a magazine article.

The book is written using the tried and true method of *linear exposition*—a form that has been used since writing was invented. The Web may change this, but it is hard to imagine precisely "how". Nonetheless, it is obvious that a web version of this book offers many more options....

[2]But footnotes provide succinct clarification that can usually be skipped by most readers.

Acknowledgments

The students in ELE 428 have had to endure a text book that is "under construction". I am am thankful for their patience and comments. I have also relied heavily on their attention when teaching the course to point out errors in draft versions of the text and areas where more explanation or examples would be useful.

Thanks to Luis Fernandes, Sophie Quigley, Jason Naughton, Cenk Bilgen, Tamer Rabie and Reza Sedagha for their helpful comments.

And, of course, to W and C.

Part I

Fundamentals

Version 1.1 Jan 24, 2003

Chapter 1

Algorithms

Algorithms and data structures are the heart and soul of understanding, formulating, analyzing and implementing computer programs to solve problems. This chapter gives a whirlwind tour of algorithms—the "heart" of programs.

People can solve problems, but that does not mean that they can express what they did as an algorithm.

As a simple example, most people can be given a a small number (say, less than 10) playing cards and re-arrange them so that they are in sorted order. Indeed, they can probably do so with about as many or even fewer movements than the best algorithms to perform the same task. Despite humans' virtuosity, they may be flummoxed when asked to describe what they did in sufficiently general terms to apply the method to arbitrarily large collections of cards; the steps also have to be described in sufficient detail so that an "unintelligent" robot could carry them out. If you can describe your method for sorting cards in a general way using only simple operations then you have formulated an algorithm.

In this chapter, we describe the nature of algorithms. We use a simple example—the problem of "sorting cards"—to explore different algorithms for solving the same problem and comparing their effectiveness.

We also examine how the algorithms can be implemented and tested using the C programming language.

Version 1.1 (2003-03-11) (Chapter version: 2002-01-07)

1.1 What is an algorithm?

A book called "Engineering Algorithms and Data Structures" surely requires that the ideas "algorithm" and "data structure" be defined. Both concepts, as well as their abstract underpinnings and some concrete implementations, will be given clear descriptions in due course. We start, however, with an intuitive definition of an algorithm.

An algorithm is:

A systematic method for solving a problem.

This deceptively simple definition requires additional clarification: what is a "solution", what is a "problem", and what is a "systematic method"? Auxiliary questions include "Should an algorithm be given a name?" and "How should it be described?" And, of course, there is the question of how "good" it is.

For the moment, we will assume that the concepts of "problem" and "solution" have intuitive meanings. Let's narrow our focus to the question: "What is a systematic method?"

All algorithms in this book will be described as Step-By-Step procedures: you always start at Step 1 and then normally continue to Step 2, then Step 3 and so on. Each step is described in English. At least one step must contain the word STOP (to tell us when—you guessed it—we should stop performing the algorithm because the solution has been found). A step is also permitted to change the default sequence; for example, although "Step 5" is normally followed by "Step 6", "Step 5" may say "Go back to Step 2" to modify the usual sequence. Indeed, we may precede an action like "go to Step i" with a conditional such as "If your mother's middle name is 'Ledgerwood', then go to Step 2." (If the condition is false, the next step is performed.)

As a trivial example, consider the algorithm to calculate the sum of an arbitrary collection of numbers:

CalculateTotal Algorithm

Calculate and output the total of n *numbers (the input)*

Step 1: Set *total* $\leftarrow 0$.

Step 2: If there are no more numbers to read, output *total* (the answer) and STOP.

Step 3: Read the next input number and add it to *total*.

Step 4: Go back to Step 2.

This example shows some other features of algorithms as presented in this book. Algorithms have a name (in this case, *CalculateTotal*) and a brief description. The description also names some parameters that reflect the size of the problem. Here the problem's size is defined by the single parameter n: how many numbers are in the collection. Algorithms perform operations on their *inputs* (here, the "collection of numbers") and produce some result as their *output* (here, the *total*). The steps are numbered sequentially and well-defined.

Donald Knuth—arguably the most influential computer scientist since the advent of computers—describes five criteria that algorithms must meet[Knu97b, p. 4–6] which we paraphrase as:

Finiteness: An algorithm must terminate in finite time.

Definiteness: Each step must be precisely defined.

Input: There are zero or more inputs.

Output: There are one or more outputs.

Effectiveness: Each step in the algorithm must be sufficiently basic that an ordinary human can perform it in finite time using operations that are "well understood" by most humans.

In many cases, a stricter definition of *effectiveness* will be used: a step is *elementary* if there is an upper bound on the time required to do it that is independent of the size parameters of the problem; otherwise, it is non-elementary.

Version 1.1 (2003-03-11) (Chapter version: 2002-01-07)

1.2 A simple sort algorithm (Selection Sort)

Suppose you have a bunch of cards with numbers written on them. You want to sort them. More specifically, you have to remove cards from the unsorted pile and create a new sorted pile with the the smallest one at the bottom.

A simple method for doing this is:

SelectionSort Algorithm

Sort n *cards*

Step 1: If there are no cards to sort, then STOP.

Step 2: Otherwise, find the smallest card, remove it and place it on top of the sorted card pile.

Step 3: Go back to step 1.

How hard is it to sort n cards using this algorithm?

We use the word "hard" here to mean the number of elementary steps that must be performed before the process stops. In this case, the number of steps is dependent on the number of cards to be sorted. For example, if there are no cards, we still have to perform Step 1. If there is one card, we have to perform Step 1, Step 2, Step 3 and Step 1 again for a total of 4 steps. In general, the total number of steps to sort n cards is $3n + 1$.

Alas, there is a problem. The three steps in the algorithm are not all elementary. Steps 1 and 3 are elementary, but Step 2 ("find the minimum") is not: the time required to perform it depends on the number of cards involved.

We could express Step 2 ("find the minimum") as a sequence of elementary operations—i.e. as an algorithm; indeed, we leave this as an exercise.

Clearly, to find the minimum in n cards, you *must* examine each one. (If you don't look at them all, there is always the possibility that one of the unexamined cards is smaller than all of the previous ones.)

We can informally describe the process of sorting 5 cards in terms of "how many cards we have to look at" as follows:

1. Look at **5** cards and put the smallest one on the table.

2. Look at **4** cards and put the smallest one on the table.
3. Look at **3** cards and put the smallest one on the table.
4. Look at **2** cards and put the smallest one on the table.
5. Look at **1** card and put it on the table.
6. STOP.

We need to look at $5+4+3+2+1 = 15$ cards
In general, we have:

$$\text{Number of cards to look at} = \sum_{0 \le i \le n} i = n(n+1)/2 = n^2/2 + n/2 \qquad (1.1)$$

and this is the total number of elementary "look-at" steps that are needed to perform all the *Step 2*s.

So we can say:

$$\text{Total number of steps} = (n+1)S_1 + \frac{n(n+1)}{2} S_{look-at} + nS_3 \qquad (1.2)$$

where S_1 and S_3 represent Steps 1 and 3 respectively.
Letting:

$$\begin{aligned} T_1 &= \text{time for Step 1} \\ T_2 &= \text{time for look-at Step} \\ T_3 &= \text{time for Step 3} \\ T(n) &= \text{time to do algorithm for } n \text{ cards} \end{aligned}$$

we obtain:

$$T(n) = (n+1)T_1 + \frac{n(n+1)}{2} T_2 + nT_3 \qquad (1.3)$$

or:

$$T(n) = (T_2/2)n^2 + (T_1 + T_2/2 + T_3)n + T_1 = c_2 n^2 + c_1 n + c_0 \qquad (1.4)$$

Version 1.1 (2003-03-11) (Chapter version: 2002-01-07)

Irrespective of the values of c_2, c_1, and c_0, the quadratic term (c_2n^2) will dominate for sufficiently large n. Indeed, the general form of the most rapidly growing term in $T(n)$ (here, the quadratic term) is all we need to know about an algorithm's performance for large problem sizes. In the case of Selection Sort, the preceding analysis produces our jewel:

Selection sort is a quadratic algorithm

Note that this characterization ignores the specific values of the constants c_2, c_1 and c_0. Nonetheless, we can still calculate reasonably accurate numbers about the behavior of the algorithm given only its quadratic nature and a single performance time.

For example, suppose that the algorithm is encoded into some programming language and executed on some machine with an input of 100,000 cards. We measure the time to perform the sort as 10 seconds. How long will the same implementation of the algorithm take to sort 200,000 cards (twice as many)?

Assuming a problem size of 100,000 is "sufficiently large" to use the quadratic approximation, doubling the size of the problem quadruples the amount of time; hence, it will take about 40 seconds to sort 200,000 cards.

Let's replay that reasoning a bit more slowly. If the algorithm is quadratic, the time to perform the algorithm for input size n is:

$$T(n) = c_2n^2 + c_1n + c_0 \quad \text{(where } c_2\text{, } c_1 \text{ and } c_0 \text{ are constants)}$$

But for "large" values of n, the fastest growing term dominates and $T(n) \approx c_2n^2$. Since we know that $T(100000) = 10$, we can calculate c_2 as $c_2 \approx 10/(10^5)^2 = 10^{-9}$. Hence, $T(200000) \approx 10^{-9} \times 200000^2 = 10^{-9} \times 4 \times 10^{10} = 40$.

Calculating the value of c_2 was unnecessary, however, since we could simply reason: $T(2n) = c_2(2n)^2 = 4 \times c_2n^2 = 4T(n)$. In short, (as stated previously) doubling the size of a quadratic algorithm quadruples the time.

Had the algorithm been *linear*, it would take 20 seconds to solve the larger problem; had it been *cubic*, it would take 80 seconds; and, had it been *logarithmic*, it would take 10.6 seconds to sort 200,000 cards.

The statement about the logarithmic performance may benefit from more detailed examination as outlined below:

$$T(n) \quad = \quad k \log n$$

$$\begin{aligned} 10 &= k \log 100000 \quad \text{(the known case)} \\ \Rightarrow k &= 10/\log_{10} 10^5 = 10/5 = 2 \end{aligned}$$

Hence:

$$\begin{aligned} T(200000) &= 2 \log_{10} 200000 \\ &= 2(\log_{10} 100000 + \log_{10} 2) \\ &= 2(5 + 0.3) \\ &= 10.6 \end{aligned}$$

> Some readers may wonder why we used base-10 logarithms instead of natural logs or base-2 logs or something else. After all, the statement of the problem did not specify the base of the logarithms. In fact, you would get the same result no matter what base was used; base-10 logs just made life simpler. Problem 1.11 (and its answer) explores this.

In our future analyses of various sorting algorithms, we will evaluate their performance in terms of the number of *compare*, *move*, and *exchange* operations they perform to sort n cards. Indeed, it is usually the case that the number of comparisons alone gives a good estimate of the relative effectiveness of different sort algorithms. (We will justify this simplification in Chapter 4.)

1.3 A better sorting algorithm (Merge Sort)

In this section we will describe a superior sort algorithm—*MergeSort*—which is not too difficult to understand. We will begin with a related algorithm—merging; next we describe the merge sort algorithm; finally, we will analyze its performance and prove that it is much better than selection sort.

1.3.1 Merge

First, we examine the problem of merging two sorted decks and then how this technique can be used to implement the Merge Sort algorithm. The "merge" problem can be described as follows.

Suppose we have two piles of sorted cards. How do we create a single pile that is sorted?

The basic algorithm is:

Merge Algorithm

Merge two sorted piles into a single sorted pile of n *cards*

Step 1: If both piles are empty, STOP.

Step 2: If the left pile is empty, place the entire right pile face down on top of the sorted pile and STOP.

Step 3: Otherwise, if the right pile is empty, place the entire left pile face down on top of the sorted pile and STOP.

Step 4: Otherwise, look at the top of each pile, choose the biggest, and place it face down on top of the sorted pile.

Step 5: Go back to Step 1.

In even simpler terms, you just repeat Step 4 ("pick the biggest from the tops of the two piles"). In the worst case, you have to do n comparisons. Hence, the *merge* algorithm is *linear* (i.e. the total number of elementary steps to perform the algorithm is of the form $c_1 n + c_0$.)

1.3.2 Combining Selection Sort and Merging

Suppose we wish to sort a pile of 16 cards. Using Selection Sort would require $16 \times 15/2 = 120$ comparisons[1]. However, if we divide the pile into two equal piles of 8 cards each, we can sort each pile independently with $8 \times 7/2 = 28$ comparisons. Since we have to do this twice, we need 56 comparisons in all to get two sorted piles of 8 cards each.

We can then merge the two sorted piles into a single one with, at worst, an additional 16 comparisons. The total number of comparisons, then, is $28 + 28 + 16 = 72$ which is quite a bit better than than the 120 required with the pure Selection Sort algorithm. We can express this idea as the following algorithm.

[1]Note that the number of comparisons is $\sum n(n-1)/2$ which differs from the number of "look-at" steps because finding the minimum of n items requires $n - 1$ comparisons, not n "look-ats".

MergeSelSort Algorithm

Sort n *cards, where for simplicity* n *is even*

Step 1: Split the deck into 2 piles (of equal size).

Step 2: Sort the left pile using SelectionSort.

Step 3: Sort the right pile using SelectionSort.

Step 4: Merge both piles into the final pile.

It is debatable whether this is a better algorithm. While it is true that fewer comparisons are required, it is also true that the algorithm is more complex. Furthermore, it is still a quadratic algorithm (one of the problems asks you to prove this); at best, the values of some of the constants c_2, c_1 or c_0 may be smaller.

1.3.3 The Merge Sort algorithm

This basic idea can be applied over and over again. Thus, instead of using Selection Sort for each pile of 8 cards, we could split each into 2 piles of 4 cards and so on.

Author's note: A diagram would be useful here.

MergeSort Algorithm

Sort n *cards*

Step 1: If there are no cards or only 1 card, STOP.

Step 2: Otherwise, split the deck into 2 piles (of about equal size).

Step 3: Solve the problem again using only the first pile.

Step 4: Solve the problem again using only the second pile.

Step 5: Merge both piles into the final pile.

We can express the idea as simply "Split the deck in 2; sort each sub-deck; merge the results".

Note that the easiest way to implement this algorithm in a programming language is to use recursion. You have probably already learned about recursion; we will also review this important concept in Chapter 2.

1.3.4 The analysis of "merge sort"

How can we determine the time required to perform this algorithm?

We start with the basic equation:

$$\begin{aligned}\text{Time to sort } n = &\text{ Time to split deck}\\ &+ \text{Time to sort left pile (size} = n/2)\\ &+ \text{Time to sort right pile (size} = n/2)\\ &+ \text{Time to merge piles (total size} = n/2 + n/2 = n)\end{aligned}$$

We can simplify this equation with the following conventions and assumptions:

- Let $T(n)$ represent the time to sort n cards.
- Let T_{split} be the time to split a deck.
- Let $K_1 n + K_0$ be the time to merge n cards (using our previous insight that merging is a linear algorithm).

We obtain:

$$T(n) = T_{split} + T(n/2) + T(n/2) + K_1 n + K_0 \tag{1.5}$$

We now make some broad simplifications to this equation. First, we will assume that $T_{split} = 0$ and $K_0 = 0$; second, we will set $K_1 = 1$. These assumptions will make our lives easier, but are they valid? Our goal is to determine the most rapidly growing term in $T(n)$ and, intuitively, it seems reasonable to set constants to either zero (when we think they will not be significant in the most rapidly growing term) or one (when we think they will). Chapter 4 grounds these simplifying assumptions on a firm mathematical basis.

With these simplifications, we obtain:

$$T(n) = 2T(n/2) + n \tag{1.6}$$

This kind of equation is called a *recurrence*—i.e. an equation that expresses its value in terms of its values for smaller arguments. The value of

one or more base cases must also be given[2]; recurrences can be then solved by hand in a "bottom-up" manner for small values. For example, suppose that we are given $T(1) = 0$ for the above recurrence. We can then calculate $T(2)$ directly from the recurrence as: $T(2) = 2T(2/2) + 2 = 2$. Table 1.1 below calculates $T(n)$ for $n = 2^i, i = 1, 2, 3, 4, 5, 6$.

n	$T(n/2)$	$2T(n/2) + n$
2	0	$2 \times 0 + 2 = 2$
4	2	$2 \times 2 + 4 = 8$
8	8	$2 \times 8 + 8 = 24$
16	24	$2 \times 24 + 16 = 64$
32	64	$2 \times 64 + 32 = 160$
64	160	$2 \times 160 + 64 = 384$

Table 1.1: $2T(n/2) + n$ for selected values

We would like to obtain a closed form solution for $T(n)$. Table 1.2 compares $T(n)$ with the simple functions $\lg n$, n and n^2. (Note that we use "lg" to indicate logarithms of base 2 (i.e. $\log_2$), "ln" for natural logs, "$\log_{10}$" for logs of base 10 and "log" for logarithms of undefined base.)

We see plainly that $T(n)$ grows faster than $\lg n$ or n, but more slowly than n^2. Hence it grows at some rate between linear and quadratic. Perhaps it grows as $n^{1.5}$, although if you calculate this function it will become apparent that $T(n)$ grows more slowly[3] than $n^{1.5}$.

If you examine the table closely, however, you may note that multiplying column 1 ($\lg n$) with column 2 (n) produces column 3 ($T(n)$). Even if you add more rows to the table, you will find that, in general, $T(n) = n \lg n$.

How do we prove that $T(n) = n \lg n$?

We use *mathematical induction*.

To use mathematical induction, we must show that the statement we wish to prove is true for one or more *base cases* and then show that if it is true for smaller numbers, it is also true for larger numbers. For example, if the statement $f(n) = g(n)$ is known to be true for $n = 0$ (the base case) and if

[2]There is some similarity between recurrences and differential equations. In both cases, an exact, closed form solution cannot be found unless there are BCs. For differential equations, a BC is a boundary condition; for recurrences, it is a base case.

[3]A problem in Chapter 4 asks you to prove that $n \lg n$ grows slower than $n^{1+\epsilon}$, $\epsilon > 0$.

$\lg n$	n	$T(n)$	n^2
1	2	2	4
2	4	8	16
3	8	24	64
4	16	64	256
5	32	160	1024
6	64	384	4096

Table 1.2: $T(n)$ compared with other functions

we can prove that if it is true for $n = k$, it can be shown that it is also true for $n = k + 1$, then it must be true for all n.

In the present case, we are told that $T(1) = 0$ and our table shows that the general formula also holds for $n = 2$ (our base case).

Let us assume that $T(n) = n \lg n$ and prove that this implies $T(2n) = 2n \lg 2n$.

We know that:

$$T(2n) = 2T(n) + 2n$$

But, since $T(n) = n \lg n$ (our hypothesis), we have:

$$T(2n) = 2n \lg n + 2n$$

or:

$$T(2n) = 2n(\lg n + 1)$$

But, since $\lg 2 = 1$, we can write:

$$T(2n) = 2n(\lg n + \lg 2)$$

And, finally, since $\log x + \log y = \log xy$; we can write:

$$T(2n) = 2n \lg 2n$$

This completes the proof that the recurrence $T(n) = 2T(n/2) + n$ where $T(1) = 0$ is solved in closed form with the formula $T(n) = n \lg n$.[4]

[4] OK... We only really proved the formula for n's that are exact powers of 2. Even more precisely, we assumed n was of the form 2^i and did mathematical induction on i.

1.4 Implementation

The previous section concentrated on the description and analysis of simple algorithms without reference to their implementation in any particular programming language or how the data they operate on is represented in a computer. This section deals with these ideas in general and how they relate to the C programming language in particular.

The sorting algorithms we have discussed have been phrased as sorting "a pile of cards". This metaphor allows you to perform the algorithm manually (or as a thought experiment) so that you can get a good intuitive feel for the algorithm. But it is not an appropriate metaphor for writing a computer program.

Our "pile of cards" is a concrete instance of a more abstract concept: "a collection of things". Collections are most commonly and easily implemented in programming languages as either *arrays* or *lists*. The C programming language, like many others, directly supports arrays and this is the way we implement our collection.

Furthermore, we need some specific kind of object to sort. We choose to sort `int`s. (It would be nice to be able to sort any kind of data with a single sort routine, but we defer a discussion on the ways to do this until Chapter 5.) We can make the algorithm somewhat more general, however, by specifying that it can sort any sub-array (from index `first` to `last` inclusively) of the array. The Selection Sort algorithm to sort a sub-array of `int`s is:

SelectionSortArray Algorithm

Sort a sub-array of n *elements from* first *to* last *inclusively*

Step 1: If the sub-array to be sorted contains no elements (i.e. if the first index is $\geq$ the last index), STOP.

Step 2: Otherwise, interchange array element `array[first]` with the smallest value in the sub-array `array[first+1]..array[last]`.

Step 3: Set *first* $\leftarrow$ *first* $+ 1$.

Step 4: Go back to *Step 1*

Version 1.1 (2003-03-11) (Chapter version: 2002-01-07)

Before writing the C code, note that an algorithm whose last step goes back to a conditional step is semantically equivalent to a `while` block, except that the test in the `while` statement is the opposite of the one in the `if` statement.

> Let's explore this "semantic equivalence" more closely. A generic algorithm can be:
>
> ***Step i:*** If *foo is TRUE*, STOP.
>
> ***Step i+...*** Other steps.
>
> ***Step k:*** Go back to *Step i*.
>
> The mechanical translation of this kind of step-by-step sequence to pseudo C is (where we assume that STOP means `return`:
>
> ```
> step_i: if (/* "foo" is TRUE*/)
>             return;
>      /* ... other steps ... */
> step_k: goto step_i;
> ```
>
> This is then converted to a `while` loop as follows:
>
> ```
> while (/* "foo" is FALSE */)
> {
>      /* ... other steps ... */
> }
> return;
> ```

The C implementation is trivial to write:

```
void mySort(int array[], unsigned int first, unsigned int last)
{
  int i;
  /* Step 1: Is there nothing to sort? */
  while (first < last)
     /* Step 2: Swap... */
     for(i = first+1; i <= last; i++) {
       /* Find smallest one in rest of array */
```

```
      if(array[first] >  array[i])) {
        /Step 2..continued...swap them */
        int tmp;
        tmp = array[first]
        array[first] = array[i];
        array[i] = tmp;
     }
    first++;
}
return;
```

1.4.1 A test harness for implementation

We will now re-write the C implementation of the selection sort algorithm using a "test harness" environment that will allow us to measure and compare the effectiveness of different sorting algorithms.

We first note (as stated previously) that all sorting algorithms (except those which do not use pair-wise comparisons, such as radix sort) involve the use of comparisons between two elements and possibly the interchange of two elements or the copying of an element to some auxiliary structure.

If we analyze Selection Sort, for example, we see that the number of comparisons required is always:

$$\sum_{1 \leq k \leq n-1} i = n(n-1)/2$$

The algorithm performs an interchange of elements (a swap operation) only when a comparison finds a smaller number. Consequently, the number of swap operations can be no larger than the number of comparisons. The worst case (i.e. number of swaps = number of comparisons) occurs when the input is in reverse sorted order. The best case, no swaps at all, occurs when the input is already sorted.

This theoretical analysis is summarized in Table 1.3.

While this analysis is, in fact, correct, it would be useful to actually confirm it with a C implementation of the algorithm that tracks how often each of the operations are performed and reports the results. This would allow us to confirm our theory with practice.

We adopt the convention that we will *not* directly compare two elements; rather, we will invoke a function `myCompare(int e1, int e2)` to compare

Operation	How often	Notes
compares	$n(n-1)/2$	Independent of input order.
swaps	$\geq 0, \leq n(n-1)/2$	Best: input sorted; worst: reversed
copies	0	Copy operation never used.

Table 1.3: Analysis of selection sort

two elements. Similarly, we will interchange elements with `mySwap()` and copy them with `myCopy()`.

In particular, we will assume a module[5] called *metrics* exists with the following Application Programming Interface (API):

int myCompare(int i, int j): Compares its arguments, i and j, returning:

$$\text{return value} = \begin{cases} < 0 & \text{if } i < j \\ 0 & \text{if } i = j \\ > 0 & \text{if } i > j \end{cases}$$

The function keeps track of how many times it is invoked.

void mySwap(int * ip1, int * ip2): Swaps the integers pointed to by its two arguments.

The function keeps track of how many times it is invoked.

void myCopy(const int * from, int * to): Copies the integer pointed to by its first argument to the location indicated by the second argument.

The function keeps track of how many times it is invoked.

unsigned int getNumCompares(): Returns the number of times `myCompare()` was invoked.

unsigned int getNumSwaps(): Returns the number of times `mySwap()` was invoked.

unsigned int getNumCopies(): Returns the number of times `myCopy()` was invoked.

[5] We use the word *module* here to mean a single object file that contains several related functions that have a well-defined interface.

Using this API[6], we can re-write the selection sort C implementation as:

```
void mySort(int array[], unsigned int first, unsigned int last)
{
  int i;
  /* Step 1: Is there nothing to sort? */
  while (first < last) {
     /* Step 2: Swap... */
     for(i = first+1; i <= last; i++)
         if(myCompare(array[first], array[i]) > 0)
             mySwap(&array[first], &array[i]);
     first++;
  }
  return;
}
```

(The complete source code is given in Appendix E or in the file `src/algorithms/selectionSort.c`. Note that to use the metrics module, the header file `metrics.h` must be included in the source code and the object file `metrics.o` must be linked with the application.)

We can now write a main "driver" program to exercise the algorithm and print statistics as follows:

```
#include "metrics.h"
#define MAX_SIZE 1000000
int main(int argc, char * argv[])
{
  int i, a[MAX_SIZE];

  /* Read ints from stdin into an array */
  for(i = 0; (scanf("%d", &a[i]) != EOF) && (i<MAX_SIZE); i++)
    ;

  /* sort the array */
  mySort(a, 0, i-1);
```

[6]You do not need to know how these functions work in order to use them. Should you be interested, however, they can be found in the files `metrics.c` and `metrics.h` in the directory `src/algorithms`.

```
  /* Print stats */
  fprintf(stderr, "Comparisons: %d\n", getNumCompares());
}
```

(The complete source code is given in Appendix E or in the file `src/algorithms/sortDriver.c`.)

We can then create an executable file such as `selSort` by linking the object codes for `sortDriver.o` (which provides "main"), `selectionSort.o` (which provides an implementation of "mySort") and `metrics.o` (which provides the copy, compare and swap functions) together.

This approach has the additional advantage that we can use the same main routine with different implementations of the `mySort` function. For example, if we implemented the Merge Sort algorithm in a file `mergeSort.c`, then we could create an executable "mergeSort" by linking `mergeSort.o`, `sortDriver.o` and `metrics.o`.

1.5 Further reading

Knuth[Knu97b, p. 7–9] gives a more mathematically precise definition of an algorithm using set and language theory. This more formal definition is important in understanding more advanced concepts in algorithm theory such as NP-complete problems.

Corman et al.[THC90, p. 2–4] use pseudo-code rather than step-by-step English descriptions to describe algorithms.

Kernighan and Pike[KP99, p. 33] give an excellent and succinct explanation and implementation of quicksort.

The idea for a sort test harness was borrowed from *The Java Programming Language*[GA97].

1.6 Problems

1.1 Formulate the "find minimum" algorithm required in the selection sort algorithm.

1.2 Suppose there is a collection of cards and each card has a single integer printed on it. Describe a method for determining if the collection contains

any duplicate cards. What is the complexity of your algorithm as a function of n, the total number of cards in the collection. (Try to find a simple $n \log n$ algorithm.)

1.3 Given a sorted collection of 1023 numbered cards, what is the smallest number of cards you have to look at to determine if a certain number is in the collection or not.

Describe a general algorithm to determine if a particular item is in a sorted collection of n objects. (You may assume that the number of cards is of the form $2^i - 1$.) What is the complexity as a function of n of your algorithm?

1.4 In our analysis of the running time of merge sort, we assumed $T(1) = 0$ and the time to merge n cards was n.

1. Suppose $T(1) = 1$. Calculate the first few values of $T(2^i)$ where $i = 0, 1, 2, 3, \ldots$. Guess the closed form solution for $T(n)$ and prove it.

2. Repeat the above problem assuming $T(1) = 0$ (the original assumption) but that the time to merge n cards is $2n$. (i.e. the recurrence is now $T(n) = 2T(n/2) + 2n$).

3. Repeat again assuming that $T(1) = a$ and $T(n) = 2T(n/2) + bn$.

1.5 Define an algorithm called CalculateAverage that outputs the average of a collection of n numbers.

1.6 Insertion Sort is informally described (for a pile of cards) as: consider the deck to be divided into an initial sorted part (at the beginning this "sorted part" is empty) followed by an unsorted part. Take each of the cards in the unsorted part in turn and insert them into the sorted part at the proper position.

Express this as an algorithm using piles of cards.

Express it as an algorithm for sorting a sub-array of `ints`.

Implement it in C. How many comparisons, swaps and copies are required in the worst and best cases?

1.7 Bubble Sort is informally described as going through an array one pair of elements at a time. (Note that the pairs overlap. For example, if the first pair is "element 1" and "element 2", then the next pair is "element 2" and

"element 3".) Whenever the pair is unsorted, their positions are switched. You keep on going through the array in this way until the whole thing is sorted.

Express this as an algorithm using piles of cards.

Express it as an algorithm for sorting a sub-array of `int`s.

Implement it in C. Analyze its worst and best case complexity (i.e. number of swaps, copies, and compares).

1.8 How many steps are required to determine the total of n items using the CalculateTotal algorithm? If Step i takes a constant time, T_i, to perform, derive an equation for the time required to do the algorithm for n inputs. Characterize the complexity of the algorithm (e.g. quadratic, cubic, linear, logarithmic, or ...)

1.9 Suppose an algorithm takes 5 seconds when the input size is 20,000. How long does it take if the input size is 100,000 under each of the following conditions:

- The algorithm is linear.
- The algorithm is logarithmic
- The algorithm is cubic.
- The algorithm is $n \log n$.
- The algorithm is quadratic.
- The algorithm takes constant time.

1.10 You will need some equipment for this problem: a deck of playing cards (52 cards, no Jokers) and a stopwatch. Shuffle the deck, take some cards from it, and time how long it takes to sort them using different methods.

You may use any sort criteria you wish, but I suggest you treat all ♠ cards > ♡ cards > ♢ cards > ♣ cards. For example, sorting the cards 4♣, 7♡, 2♠, 3♡ would result in 2♠, 7♡, 3♡, 4♣.

In particular, try the following:

- Select 8 cards at random and sort them using whatever method comes "naturally" to you. (You are allowed to look at all the cards at once and use your innate "parallel processing" abilities if you wish.) How long does it take? (You may wish to try it a few times and take the average.)

- Select 8 cards at random and sort them by mechanically following the SelectionSort algorithm. (In this case, you are only allowed to look at one or two cards at a time. Pretend you are a robot.) How long does it take?
- Do it again (as a "robot") using the MergeSort algorithm. How long does it take.
- Repeat all of the above for 16 and 32 cards. Does you experiment confirm the mathematical analysis of the complexity of the two algorithms?

1.11 The text states that the base of logarithms is irrelevant when talking about an algorithm being of logarithmic complexity(page 9). Explain why.

1.12 Formulate the selection sort algorithm for a computer implementation that uses lists instead of arrays.

1.13 Knuth describes an algorithm for calculating the date of Easter[7] for any year after 1582 (A.D.) as follows[8].

Easter Algorithm

Determine date of Easter for the year Y

Step 1: Set $G \leftarrow (Y \bmod 19) + 1$.

Step 2: Set $C \leftarrow \lfloor Y/100 \rfloor + 1$.

Step 3: Set $X \leftarrow \lfloor 3C/4 \rfloor - 12$, $Z \leftarrow \lfloor (8C+5)/25 \rfloor - 5$.

Step 4: Set $D \leftarrow \lfloor 5Y/4 \rfloor - X - 10$.

[7] Knuth has stated that there "are many indications that the sole important application of arithmetic in Europe during the Middle Ages was the calculation of [the Easter date]". A delightful article in *The Sciences*[Hay99] describes the astounding mechanical inventiveness of the Strasbourg Clock creators who implemented this algorithm and others with gears and springs hundreds of years ago. The huge clock—over 5 stories high—does not suffer from the Y2K bug. (A minor modification will be required after the year A.D. 9999, but it will only cost a few cents not trillions of dollars!)

[8] The notation $\lfloor x \rfloor$ means the biggest integer $\leq x$.

Step 5: Set $E \leftarrow (11G + 20 + Z - X) \bmod 30$. If $E = 25$ and $G > 11$, or if $E = 24$, then increase E by 1.

Step 6: Set $N \leftarrow 44 - E$. If $N < 21$, then set $N \leftarrow N + 30$.

Step 7: Set $N \leftarrow N + 7 - ((D + N) \bmod 7)$.

Step 8: If $N > 31$, the date is $(N - 31)$ `April`; otherwise the date is N `March`. STOP.

a) Are any of the steps non-elementary?

b) Can the algorithm in its entirety be considered to be an "elementary" step?

c) Implement it in C.

d) Is there any software you use regularly that implements this algorithm? If yes, and if the source code is available, look at it for other date examples.

e) What's so special about the year 1582?

f) Do you understand why the algorithm works? Does it matter?

1.14 Prove that the *MergeSelSort* algorithm (page 10) is quadratic.

1.15 Suppose you want to modify an array of integers so that all of the elements in the array from index 0 to index $n - 1$ are shifted upwards by one and that the original value in `array[n]` is placed in `array[0]` For example, if the array were |3|7|9|2|... and n were 3, the array would become: |2|3|7|9|...

1. Formulate an algorithm to do this using only copy operations and implement using the `myCopy` function in the metrics module.

2. Formulate an algorithm to do this using only swap operations and implement using the `mySwap` function in the metrics module.

3. What is the complexity of each implementation? Which one would be faster?

4. What ANSI C standard library function would solve the problem most efficiently?

1.16 Examine the source code for the metrics module (`metrics.c` and `metrics.h`).

1. Explain the use of the "main" function in `metrics.c`. Why does it not conflict with the "main" function in `sortDriver.c`?

2. How could the module (and its API) be modified so that attempts to swap or compare elements that were not within the array bounds could be detected?

1.17 Modify the implementation of `mySort` on page 16 to sort double precision numbers instead of integers. (Note: do not use the metrics module.)

1.18 Modify the implementation of `mySort` on page 16 to sort strings instead of integers. (Note: do not use the metrics module.)

1.19 Describe an algorithm to merge 3 piles of sorted cards into a single pile. What is the complexity of the algorithm?

1.20 Consider an algorithm similar to merge sort except that the pile is split in three, each sub-pile sorted, and then the sorted piles merged.

1. Formulate this idea as an algorithm.

2. Describe a recurrence for the time to sort n cards ($T(n)$).

3. Simplify the recurrence using methods similar to the ones used in the text for merge sort. Assume that $T(1) = 0$ and calculate $T(n)$ for $n = 1, 3, 9, 27$. Guess and prove a closed form solution.

1.21 One of the oldest known algorithms (c. 500 B.C.) is Euclid's Algorithm for determining the largest common factor of two integers. The method can be informally described as "if the smaller number is a factor of the bigger one, the answer is the smaller number; otherwise, divide the small one into the bigger one and note the remainder. Then replace the big one by the small one and the old smaller one with the remainder. Solve the problem again."

For example, the greatest common divisor (gcd) of 1769 and 551 is 29. (Specifically, $1769 = 61 \times 29$ and $551 = 19 \times 29$.) The sequence of steps in obtaining this result are:

$$\begin{aligned} 1769 \div 551 &\Rightarrow 3 \text{ (remainder } = 116) \\ 551 \div 116 &\Rightarrow 4 \text{ (remainder } = 87) \\ 116 \div 87 &\Rightarrow 1 \text{ (remainder } = 29) \\ 87 \div 29 &\Rightarrow 3 \text{ (remainder } = 0) \\ \text{Hence, } \gcd(1729, 551) &= 29 \end{aligned}$$

Express Euclid's method as an algorithm.
Implement it in C.

1.22 Suppose a program takes 1 microsecond to solve a problem of size $n = 1$. (i.e. $T(1) = 1\mu$sec) Suppose that, in general, $T(n+1)$ is 1 % more than $T(n)$.

1. Find a closed form equation for $T(n)$.
2. How long would it take to solve the problem for $n = 100$? What about $n = 3000$?

Chapter 2

Recursion

Recursion is an important concept in computer science and we will define what it means in this chapter. Many algorithms are best expressed recursively and many of the data structures we will examine later in this book are also defined recursively. Programming languages themselves—the "stuff" of algorithm implementations—are also described recursively using BNF notation (which is discussed in Chapter 3).

2.1 What is recursion?

We call something *recursive* if it is defined in terms of itself. To avoid an infinite regress, the definition must include one or more base cases that are directly defined; the recursive part of the definition must split the definition into simpler instances.

We have already seen this concept used in recurrences, the kind of equation (like $T(n) = 2T(n/2) + n$) that we developed and solved in the previous chapter.

Recursive algorithms often follow the general "Divide and Conquer" method: the problem to be solved is split (divided) into simpler problems which are then solved (conquered) using the same method. In order to work, the division and re-division of the problem must (eventually) result in one or more base cases that can be solved directly.

"Divide and conquer" recursive algorithms often follow the general pattern:

First: If the problem is simple enough to solve directly, then solve it and

STOP.

Second: Otherwise, split the problem into one or more simpler problems.

Third: Solve each of the sub-problems using this method.

Finally: If necessary, combine the solutions of the sub-problems into the solution for the original problem.

One aspect that is often encountered in recursive algorithms is the absence of any explicit loops.

2.2 Simple example: Addition

Suppose we wish to teach a child how to add two numbers. We assume that the child has learned how to count both forwards and backwards.

Even more specifically, we will ask the child to add two non-negative numbers and we will ensure that the first number—which will be written on a pink card—is no bigger than the second number which will be on a blue card. For example, we can ask the kid to add "5 (pink) and 6 (blue)" or "123(pink) and 200(blue)", but not to add "7 (pink) and 6 (blue)".

The algorithm is:

AddPinkBlue Algorithm

Add a pink number to a blue number and return answer

Step 1: If the pink number is ZERO, the answer is the blue number;STOP.

Step 2: Otherwise, count backwards once from the pink number. Erase the old number on the pink card and write in the new one.

Step 3: Count forwards once from the blue number. Replace the number on the blue card with this one.

Step 4: Solve the new problem using this method.

Using only the C language "if" construct and the primitive increment and decrement operators (i.e. "count forwards" and "count backwards"), we can implement the algorithm in C as follows:

```
unsigned int add(unsigned int pink, unsigned int blue)
{
   /* Step 1: is pink number ZERO? */
   if (pink == 0)
       return blue;

    /* Step 2: replace pink number by counting backwards */
    pink--;

    /* Step 3: replace blue number by counting forwards */
    blue++;

    /* Step 4: Solve the new problem using "this" algorithm */
    return add(pink, blue);
}
```

2.3 How does recursion work

The "recursiveness" of the previous algorithm is encapsulated with "Step 4": the algorithm does not say "go back to Step 1"; rather, it says "solve the problem again" using the new inputs.

(We will discuss the relative merits of replacing "Step 4" with the non-recursive "Go back to Step 1" shortly. For the moment we will continue to use the recursive "Step 4"—solve the problem again.)

Let's consider the problem from the point of view of children solving it. Suppose, for example, that we give Jane a pink card with "3" written on it and a blue card with "5" on it.

Jane follows Steps 1–3 of the algorithm creating new pink and blue cards with the values "2" and "6" on them. Then she gets to "Step 4" .... At this point, she has a new problem to solve—add a "pink 2" card to a "blue 6" card. This new problem can be solved by following the same rules that Jane herself has just followed.

Jane, however, has a (gullible) friend, Dick. Jane says, "Hey, Dick, I've got a blue number and a pink number—follow the rules and tell me the answer.". Dick is a complacent fellow; he takes the inputs (the values of the blue and pink cards), follows the directions, and produces new pink and blue cards just before he gets to "Step 4".

Version 1.1 (2003-03-11) (Chapter version: 2003-01-24)

At this point Dick has a new problem to solve: having reduced "pink/blue" from "2/6" to '1/7", he asks someone else, Sally, to solve the new problem.

Sally is given a pink card with a "1" and a blue card with "7" on it. She follows steps 1—3, producing new cards with a "0" and "8" on them.

She gives the new cards to Spot. Spot looks at the pink card, sees that it has a zero on, and so says the answer is "8" and stops.

Sally hears Spot give the answer "8" and says, "Great, the answer is 8." Similarly, Dick hears Sally's answer and repeats it to Jane (pretending that he figured it out all by himself.) So, eventually, Jane is informed of the right answer.

The central aspect of this tale is that Dick, Jane, Sally and Spot all followed **exactly the same rules** (i.e. the used the same algorithm), but they applied the rules to **different data** (cards). The other insight is that Jane did not need any friends to solve the original problem; after transforming the problem from adding "3 and 5" to that of adding "2 and 6", she could simply have solved the new problem again herself.

Let us now consider how a recursive function works in a programming language like C. We need to think of the function as describing the rules for manipulating data; the rules exist on their own independently of the data that is manipulated. In order for the data to exist independently, the data (or, more correctly, the computer memory for the data) must be created anew each time the function is invoked. In C, the parameters passed to a function and all variables local to the function do occupy separate memory locations each time the function is invoked[1].

For example, consider the recursive call in the `add` function given previously:

```
unsigned int add(unsigned int pink, unsigned int blue)
{
          /* .....  */
    return add(pink, blue);  /* Recursive call */
}
```

Suppose that `add(2, 3)` is invoked; by the time it gets to the recursive call, the first invocation of `add` will have changed the values of "pink" and

[1]Note that the rules, which are unchanging, could be placed in ROM (Read Only Memory); the memory area for the data would have to be RAM (Random Access Memory that can be read and written to).

"blue" to "1" and "4" respectively. Before it can return, it makes the recursive call `add(1, 4)`. This second invocation of `add` will now apply the algorithm to *its* data; indeed, it will modify its value of "pink" and "blue" to "0" and "5" before making yet another recursive call. It is essential to understand, however, that when the second invocation modifies its "pink" and "blue" data, it has absolutely no effect on the "pink: and "blue" data of other invocations of `add` including the first invocation that has not yet terminated.

2.4 Tail recursion

Some programmers eschew recursion, believing that iterative or looping algorithms are inherently more efficient. While there is some validity to this argument—recursive calls do require allocating additional memory for the passed parameters and local variables—it is also true that many modern compilers automatically eliminate some common forms of recursion and the associated overhead.

One form of recursion that can be eliminated automatically, called *tail recursion*, occurs when the last statement of a recursive function invokes itself. The `add` routine described above provides an example of a tail-recursive function.

The rule for eliminating tail recursion is very simple: replace the last recursive call with a "goto" to the beginning of the algorithm.

For example, the AddPinkBlue algorithm is tail-recursive since the last step is "Solve the new problem using this method." We can easily convert the algorithm to a non-recursive version as follows:

AddPinkBlueNonRecursive Algorithm

Add a pink number to a blue number and return answer

Step 1: If the pink number is ZERO, the answer is the blue number; STOP.

Step 2: Otherwise, count backwards once from the pink number. Erase the old number on the pink card and write in the new one.

Step 3: Count forwards once from the blue number. Replace the number on the blue card with this one.

Step 4: Go back to *Step 1.*

Note that this now results in an algorithm with a loop; the recursive version had no loops.

Let's see how this works with the tail-recursive `add` function in C. First, we re-write the function in more compact fashion:

```
unsigned int add(unsigned int pink, unsigned int blue)
{
   if (pink == 0)   return blue;
   pink--;  blue++;
     /* The last step is a recursive call...
                hence this is TAIL RECURSION */
   return add(pink, blue);
}
```

We can now mechanically replace the last recursive call with a "goto":

```
unsigned int add(unsigned int pink, unsigned int blue)
{
  start:
   if (pink == 0)   return blue;
   pink--;  blue++;
     /* Replace the last step with a goto */
   goto start;
}
```

Knowing that an "if statement" followed by "other statements" followed by a "goto" back to the "if statement" is semantically equivalent to a "while statement" (although the condition tested in the "while" statement is the exact opposite of the one originally in the "if" statement), we can write:

```
unsigned int add(unsigned int pink, unsigned int blue)
{
   while (pink != 0) {
    pink--;
    blue++;
   }
   return blue;
}
```

Finally, we can optimize away the explicit comparison as follows:

```
unsigned int add(unsigned int pink, unsigned int blue)
{
   while (pink--)
    blue++;
   return blue;
}
```

By the way, the following is simpler still.

```
unsigned int add(unsigned int pink, unsigned int blue)
{
    return pink+blue;
}
```

...but it is cheating. The idea, after all, was to express the concept of addition using only the concepts of counting backwards or forwards by one and without using the "add" operator!

2.5 Example: Fibonacci numbers

Fibonacci numbers are defined by the following recurrence:

$$Fib(n) = \begin{cases} 1 & \text{if } n = 1 \text{ or } n = 2 \\ Fib(n-1) + Fib(n-2) & \text{otherwise} \end{cases}$$

The series $Fib(1)Fib(2), Fib(3)\ldots$ is called the Fibonacci series; its first few values are: 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144 ...

We can easily write a recursive program to calculate the nth Fibonacci number with a straight forward transformation of the recurrence into a C program:

```
int fib(int n)
{
  if((n==1) || (n==2))
    return 1;
  return fib(n-1) + fib(n-2);
}
```

Version 1.1 (2003-03-11) (Chapter version: 2003-01-24)

One can get a feel for how the recursion works by drawing a *recursion tree* that shows how the function invokes other instances of itself as shown in Figure 2.1.

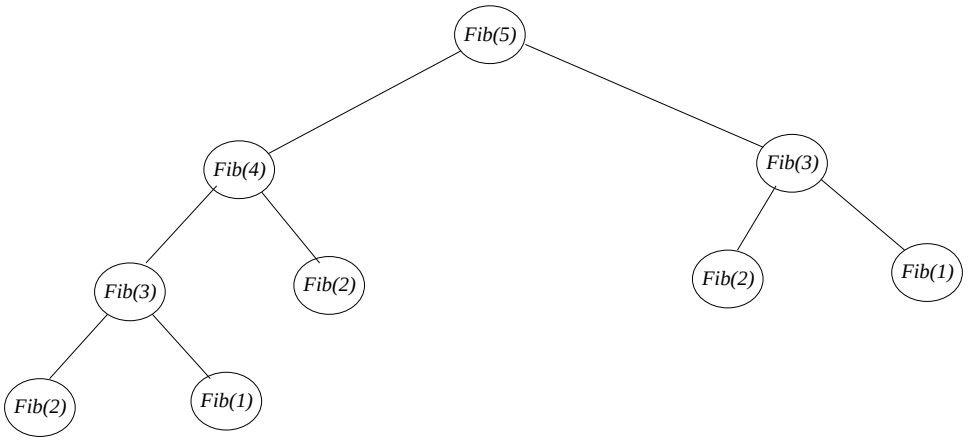


Figure 2.1: Recursion tree for $Fib(5)$

What is the complexity of this function?

First, note that each node in the recursion tree represents invoking the Fibonacci algorithm. But the time required to perform the algorithm is some constant plus the time for the recursive calls. However, all the recursive calls are in the recursion tree; consequently, the time required in each node is simply the constant overhead.

These observations lead us to conclude that the time required to solve the recurrence is proportional to the number of nodes in the recursion tree.

How many nodes are in the tree?

We can draw the recursion trees for small values of n and count the nodes. We obtain:

We note that the number of nodes for the next Fibonacci number will involve all of the nodes for the previous two Fibonacci recursion trees plus an additional root node for the new number. In short, letting $N(n)$ be the number of nodes in the recursion tree, we obtain the recurrence:

$$N(n) = \begin{cases} 1 & \text{if } n = 1 \text{ or } n = 2 \\ N(n-1) + N(n-2) + 1 & \text{otherwise} \end{cases}$$

n	*number of nodes*
1	1
2	1
3	3
4	5
5	9
6	16

This is similar to the recurrence that defines the Fibonacci numbers. Indeed, the two are closely related and it looks like:

$$N(n) = 2Fib(n) - 1$$

This can be proved by mathematical induction.

The base cases, $n = 1$ and $n = 2$ can be proven directly. The induction hypothesis is $N(n) = 2Fib(n) - 1$.

We have by definition:

$$N(n+1) = N(n) + N(n-1) + 1$$

Using our hypothesis, we obtain:

$$N(n+1) = 2Fib(n) - 1 + 2Fib(n-1) - 1 + 1 = 2(Fib(n) + 2Fib(n-1)) - 1 = 2Fib(n+1) - 1$$

It is also possible to show that[2]:

$$Fib(n) = (\phi^n - \hat{\phi}^n)/\sqrt{5}$$

or, even more simply,

$$Fib(n) = \text{ closest integer to } \phi^n/\sqrt{5}$$

where

$$\phi = (1 + \sqrt{5})/2 = 1.61803\ldots \text{ and } \hat{\phi} = (1 - \sqrt{5})/2 = -0.61803\ldots$$

[2]We will prove this in Chapter 4

Consequently, the complexity of the recursive `fib` function is exponential! You can get an intuitive understanding for this sad state of affairs when you realize that the recursive implementation involves the repeated calculation of the same Fibonacci number.

A non-recursive version can be written a simple fashion with only linear complexity. The basic idea behind a linear algorithm, is expressed as follows[3] (for the case $n > 2$):

FibonacciLinear Algorithm

Calculate the n*th Fibonacci number with linear complexity*

Step 1: Set $i \leftarrow 3$. Set $fib_1 \leftarrow fib_2 \leftarrow 1$.

Step 2: Calculate $fib_i \leftarrow fib_{i-1} + fib_{i-2}$

Step 3: If $i = n$, the answer is fib_n. STOP.

Step 4: Set $i \leftarrow i + 1$. Go back to Step 2.

We can obtain even more efficient implementation of logarithmic complexity by using the equation:

$$Fib(n) = \text{ closest integer to } \phi^n/\sqrt{5}$$

2.6 Example: Towers of Hanoi

The Towers of Hanoi problem involves three pegs made of pure diamond (according to myth) and a number of differently-sized golden disks. Initially, all the disks are neatly stacked, ordered by size with the largest at the bottom, on one of the pegs as shown in Figure 2.2.

The problem is to move the disks one at a time so they are all stacked in the same order on another peg. The only rules are that you must never have a larger disk on top of a smaller one and you can only move one disk at a time between any of the towers. Figure 2.3 below shows the steps involved for moving three disks.

[3] This is a trivial example of a technique known as *dynamic programming*.

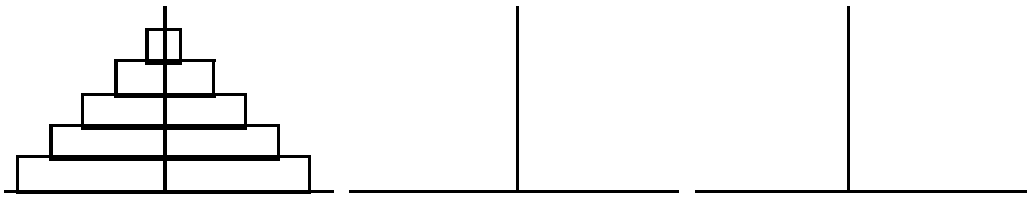


Figure 2.2: Towers of Hanoi Initial Configuration with n = 5

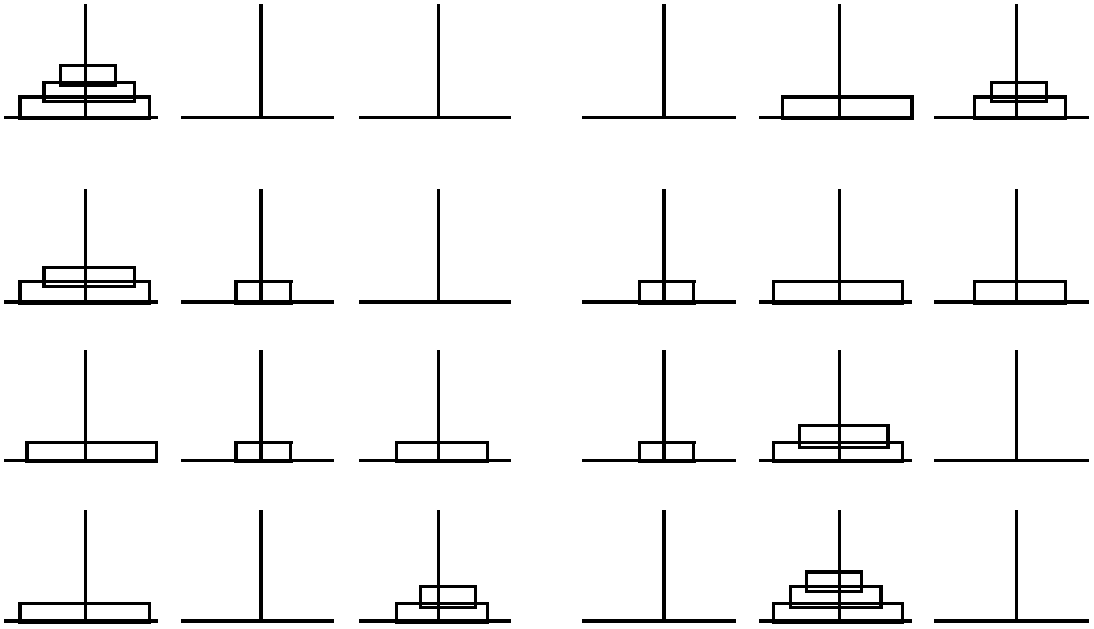


Figure 2.3: Towers of Hanoi Solution for n = 3

Suppose there were 10 disks, what sequence of moves is required to move them from peg 1 to peg 2? Adam says he knows how. But, in fact, Adam has no idea how to do it. He has a friend Betty, however, who claims to be able to solve the problem for 9 disks. Adam asks Betty to move 9 disks from peg 1 to the spare peg 3. Adam now has the simple task of moving the last disk from peg 1 to peg 2. He then asks Betty to move the 9 disks from peg 3 to peg 2 and the job is done.

When Betty said she knew how to solve the problem for 9 disks, she had in mind co-opting the services of someone who knew how to move 8 disks. Similarly, the person moving 8 disks would divide the problem up into moving 7 disks and 1 disk. In short, to move any number, n of disks, you need only know how to divide the problem and find someone who can move $n-1$ disks.

The important thing to recognize is that everyone in the chain is following the *identical* set of instructions which can be summarized as:

1. If there is only one disk to move, then move it and this solves the problem.

2. Otherwise, find someone else to move all but the bottom disk to the spare peg. (This person, of course, must respect the rules.)

3. Then move the bottom disk to the destination peg.

4. Get the other person to move all the disks on the spare peg to the destination peg. The problem is solved.

The only difference is that each person works on different data. What is the spare peg for one is the destination peg for another and so on. It is this situation of identical instructions (which include instructions to ask someone else to follow the same instructions) but different data that is the heart of recursion.

The "conceptual leap" required to formulate a good recursive algorithm involves properly dividing the problem into simpler sub-problems.

In short, the conceptual leap involved realizing that moving n disks can be done by solving the smaller problem of moving $n-1$ disks twice and moving a single disk.

We can now express the basic algorithm:

TowersOfHanoi Algorithm

Solve the Towers of Hanoi *problem for* n *disks*

Step 1: If the number of disks to move is 0 (zero), then STOP.

Step 2: Otherwise, move $n-1$ disks to the spare tower using "this" algorithm.

Step 3: Move a single disk to the destination.

Step 4: Move $n-1$ disks from the spare tower to the destination using "this" algorithm.

This algorithm can be easily expressed in C. The only slightly tricky part is figuring out how we can identify the "spare" tower. We assume that the left, middle and right towers are identified by the numbers 1, 2 and 3 respectively. We know that at any given time, exactly one of the towers is the "source", another is the "destination" and the other one is the "spare" tower. So we must have `spare` + `destination` + `spare` $= 1 + 2 + 3 = 6$. Hence, if we know two towers, we can calculate the identifying number of the third. The C code is given below:

```
/**
  * "towers" solves the Towers of Hanoi problem and writes
  * the solution to <stdout> as 2^n -1 lines in the form:
  *       <from> <to>
  *  where:
  *     <from> is the ID of a tower to pick up a disk from
  *     <to>   is the ID of where to drop the disk to
  * @param n     the number of disks to move
  * @param from  the tower ID number to move from
  * @param to    the tower ID number of the destination
  */
void towers(int n, int from, int to)
     /*
      * The standard recursive "divide and conquer" method
      * is used to solve the problem.  Specifically:
      *     1) If the number of disks to move is 0, STOP.
      *     2) Otherwise, move n-1 disks to the spare tower.
      *     3) Move a single disk to the destination.
```

Version 1.1 (2003-03-11) (Chapter version: 2003-01-24)

```
      *     4) Move n-1 disks from the spare to destination
      */
{
  if( n > 0) {
    /* Note: "spare", "from" and "to" are distinct and
     * chosen from 1 or 2 or 3.  Hence, we must have
     * the invariant:
     *     spare + from + to = 1 + 2 + 3 = 6
     */
    int spare = 6 - from - to;
    --n;
    towers(n, from, spare);
    printf("%d %d\n", from, to);
    towers(n, spare, to);
  }
}
```

How do we calculate the complexity of the algorithm? Let us call $M(n)$ the number of moves to solve the problem for n disks.

Clearly, we have the recurrence:

$$M(n) = M(n-1) + 1 + M(n-1) = 2M(n-1) + 1$$

There are many ways to solve this recurrence and some are suggested in the exercises. One way is to use a "clever" substitution, i.e. Let:

$$U(n) = M(n) + 1$$

so we obtain:

$$U(n) - 1 = 2(U(n-1) - 1) + 1 = 2U(n-1) - 1$$

or, simply,

$$U(n) = 2U(n-1) = 2(2U(n-2)) = 2(2(2U(n-3) = 2^n U(0)$$

Assuming $U(0) = 1$, we have:

$$M(n) = U(n) - 1 = 2^n - 1$$

In other words, the Towers of Hanoi algorithm is of exponential complexity. Unlike the case of the exponentially complex recursive solution to calculating Fibonacci numbers, there is no clever non-recursive solution to the Towers of Hanoi problem that does any better. It can be proven that the optimal number of moves to solve the problem for n disks is $2^n - 1$.

2.7 Example: Counting ways to make change

How do I love thee? Let me count the ways.

—Elizabeth Barrett Browning

Our last example of recursive algorithms involves calculating the total number of ways to make change using different kinds of coins.

Let us first give some simple examples of what we mean by the problem. Suppose we have pennies, nickels and dimes; how many ways can be make change of 22 cents. Let's enumerate the ways:

1 22 Pennies
2 17 Pennies and 1 Nickel
3 12 Pennies and 2 Nickels
4 7 Pennies and 3 Nickels
5 2 Pennies and 4 Nickels
6 12 Pennies and 1 Dime
7 2 Pennies and 2 Dimes
8 7 Pennies and 1 Nickel and 1 Dime
9 2 Pennies and 2 Nickels and 1 Dime

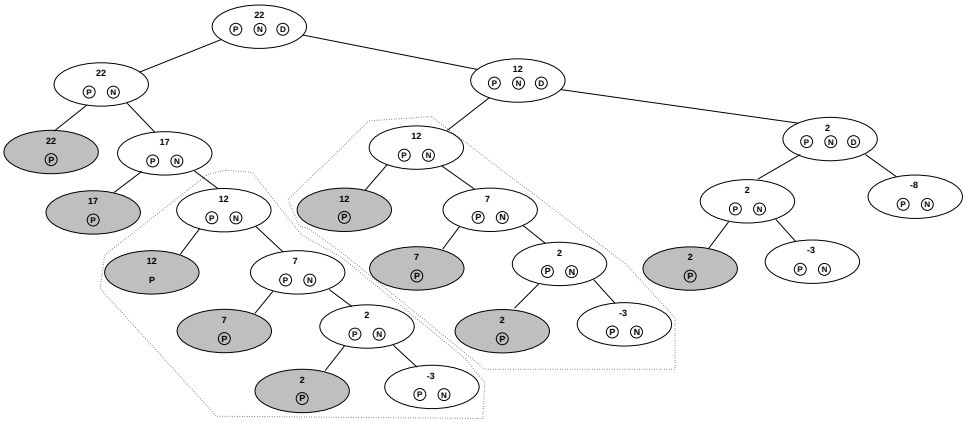


Figure 2.4: Recursion tree for changing 22 cents

The recursive solution to this problem requires the insight that solving the problem for some amount using Pennies, Nickels and Dimes can be split into two smaller problems:

1. Solve the same problem using one less type of coin (e.g. solve it using only Pennies and Nickels).

2. Solve the same problem using Pennies, Nickels and Dimes where we **know** for certain that a Dime is involved. We know this for certain if the amount to make change for is at least as big as 10 cents. Consequently, we can subtract 10 from the amount and solve the problem (which is now simpler because the amount is smaller) recursively. Of course, if after subtracting the value of the coin, we have a negative amount, we know that the coin cannot be used in the solution.

3. The answer to the original problem is the sum of the answers to the two sub-problems.

4. We escape from the recursion in three cases:

   - If the amount is negative, there is no way to make change; return 0.
   - If the number of coin types is only 1 (e.g. if only Pennies are allowed), there is exactly 1 way to make change.
   - If the amount to change is exactly zero, there is only 1 way to make change.

Figure 2.4 shows the recursion tree for calculating the number of ways to make change for 22 cents. In this figure, the amount is indicated at the top of the oval and the types of coins to use are indicated with the circled letters "P" (pennies), "N" (nickels) and "D" (dimes). Recursive calls are indicated with lines going to two lower ovals. Calls that result in a return value of 1 are shown in gray ovals.

By examining the recursion tree, we can see that the same value is calculated more than once; for example, counting the ways to make change for 12 cents using Pennies and Nickels is done twice as indicated by the sub-trees surrounded by a dotted line.[4]

[4]This suggests that there is a computationally more efficient way to do this calculation. This kind of situation is commonly solved with a technique called *dynamic programming* which we will cover in Chapter 12.

These ideas are all expressed in the following C program.

```
#include <stdio.h>
#include <stdlib.h>

/*** Typedefs ***/
typedef enum{Penny = 1,
             Nickel = 5,
             Dime = 10,
             Quarter = 25,
             HalfDollar = 50}
        Coin;

/*** Globals ***/
Coin typesOfCoins[] = {Penny,
                       Nickel,
                       Dime,
                       Quarter,
                       HalfDollar};

/*** Function prototypes ***/
int nWaysToMakeChange(int amount, int nCoinTypes);

int main(int argc, char * argv[])
{
  if ((argc != 2) || (atoi(argv[1]) < 0)) {
    fprintf(stderr, "Usage: %s amount(cents)\n", argv[0]);
    exit(1);
  }
  printf("%d\n", nWaysToMakeChange(atoi(argv[1]),
                       sizeof(typesOfCoins)/sizeof(int)));
  exit(0);
}

int nWaysToMakeChange(int amount, int nCoinTypes)
{
  if ((amount == 0) || (nCoinTypes == 1))
    return 1;
```

```
  if (amount < 0)
    return 0;
  return nWaysToMakeChange(amount, nCoinTypes-1)
    + nWaysToMakeChange(amount -
           typesOfCoins[nCoinTypes-1], nCoinTypes);
}
```

2.8 Problems

2.1 Modify `nWaysToMakeChange` so that it prints out the actual way to make change for each of the ways that it discovers.

2.2 The algorithm for adding two numbers stated that the "pink" number should be no bigger than the "blue" number. Will the algorithm still work if this rule is broken? If it does still work, what advantage, if any, is there in this rule?

2.3 What will happen if the "pink" input to the add algorithm is a negative number? Fix the algorithm so that it will work in this case.

2.4 According to one version of the "Towers of Hanoi" myth, the universe will dissolve after a tower of 64 disks has been moved. Is there anything to worry about if the task started right after the Big Bang and moves are performed at the rate of one per second? Suppose the monks are replaced by a computer that can perform a move in 1 millisecond. Is there anything to worry about? Is there anything to worry about under any circumstances? Explain.

2.5 One of the oldest known algorithms (c. 500 B.C.) is Euclid's Algorithm for determining the largest common factor of two integers. The method can be informally described as "if the smaller number is a factor of the bigger one, the answer is the smaller number; otherwise, divide the small one into the bigger one and note the remainder. Then replace the big one by the small one and the old smaller one with the remainder. Solve the problem again."

For example, the greatest common divisor (gcd) of 1769 and 551 is 29. (Specifically, $1769 = 61 \times 29$ and $551 = 19 \times 29$.)

Express Euclid's method as a recursive algorithm.

Try to determine the worst-case complexity of Euclid's algorithm as a function of the largest number. (Hint: Fibonacci numbers.)

Implement it in C.

2.6 The extended version of Euclid's algorithm not only determines $gcd(m, n)$, it also determines values a and b such that

$$am + bn = d = gcd(m, n)$$

For example, given $m = 1769$ and $n = 551$, we would obtain:

$$5 \times 1769 - 16 \times 551 = 29$$

Can you figure out a way to do this? (This problem is for mathematically-inclined readers.)

2.7 Write a non-recursive version of `fib` that is of linear complexity.

2.8 Write a non-recursive version of `fib` that is of logarithmic complexity. (Hint: The easiest way to do it is to use the approximation $Fib(n) =$ closest integer to $\phi^n/\sqrt{5}$. This version needs floating point arithmetic.)

A version using only integer arithmetic is also possible. (Hint: consider the powers of the matrix: $\left(\begin{smallmatrix} 1 & 1 \\ 0 & 1 \end{smallmatrix}\right)$.)

2.9 Modify the "towers" program so that the user can specify the towers with the words "left", "middle" and "right" instead of the numerical identifiers 1, 2 and 3. Furthermore, the output of the program should produce things like "Move a disk from the left tower to the middle one." instead of "1 2."

2.10 Solve the recurrence $M(n) = 2M(n-1)+1$ by assuming that $M(0) = 0$, calculating $M(n)$ for $n = 1, 2, 3, 4, 5$, developing a "guess" for the closed form solution and proving that the guess is correct by mathematical induction.

2.11 Define an algorithm that reverses its input (assumed to be "a pile of cards"). Express both recursive and non-recursive algorithms.

Re-formulate the algorithms for an array of integers and implement them in C.

Version 1.1 (2003-03-11) (Chapter version: 2003-01-24)

2.12 Define an algorithm to print a number in any base between 2 and 36. For bases of 10 or less use the digits 0...9; for larger bases, use the letters of the alphabet A...Z for the digits whose value is greater than 9.

For example, $123_{10} = 1111011_2(\text{binary}) = 7B_{16} = 3U_{31}$.

2.13 Ackermann's function is defined as:

$$\text{Ack}(i,j) = \begin{cases} 2^j & \text{if } i = 1 \\ \text{Ack}(i-1,2) & \text{if } j = 1,\ i > 1 \\ \text{Ack}(i-1,\text{Ack}(i,j-1)) & \text{otherwise} \end{cases}$$

Calculate Ack(2, 2) by hand using the definition. If you are very ambitious, try calculating Ack(3, 3). (This function is *highly* recursive!)

Implement it in C.

2.14 Modify the recursive version of C "pink/blue" algorithm (the `add` function on page 28) so that the arguments and results of each recursive invocation are printed to `stdout` just before the `return` statements.

2.15 Factorial n ($n!$) can be defined recursively as follows:

$$\text{fact}(n) = \begin{cases} 1 & \text{if } n = 0 \\ n \times \text{fact}(n-1) & \text{if } n > 0 \end{cases}$$

Translate the definition mechanically into a recursive function.

Is the function tail recursive?

2.16 Consider the following definition of a function:

$$\text{foo}(m,n) = \begin{cases} m & \text{if } n = 0 \\ \text{foo}(m+m, n-1) & \text{if } n > 0 \end{cases}$$

Express this function in more traditional form.

Implement the function "mechanically" as a translation of the recurrence into a recursive function.

2.17 Express a defition for the `mul` function of two non-negative integers m and n whose value is $m \times n$. Use a recurrence-like definition as was done for the factorial and "foo" functions in the previous problems. Use only addition or subtraction in the definition.

2.18 Show that the number of additions required in executing the recursive version of calculating the nth Fibonacci number is $F(n) - 1$ where $F(n)$ is the nth Fibonacci number.

2.19 Show that the number of nodes in the recursion tree for calculating the nth Fibonacci number using the recursive algorithm is $2F(n) - 1$ where $F(n)$ is the nth Fibonacci number.

2.20 Consider the following recurrence:

$$Gib(n) = \begin{cases} c & \text{if } n = 1 \text{ or } n = 2 \\ Gib(n-1) + Gib(n-2) + k & \text{otherwise} \end{cases}$$

Clearly, this becomes the Fibonacci recurrence in the special case where $c = 1$ and $k = 0$.

Find a closed-form solution for $Gib(n)$ in terms of $Fib(n)$, c and k.

2.21 Compile and run the linear and recursive C programs to calculate $Fibonacci(n)$. Try each one for $n = 20, 25, 30, 35, 40, 45$ (for example). Compare the time to run (to tenth second accuracy for each version. What are your conclusions?

Version 1.1 (2003-03-11) (Chapter version: 2003-01-24)

Chapter 3

Parsing

3.1 What is formal language theory?

We usually think of the word "language" in terms of human languages such as English or French, but we also talk of the C or Pascal programming languages. We will use the term *natural language* for the strategy that evolved for communication in human society and *formal language* for the more strictly defined languages that computers can "understand".

Our concern is with formal languages. In additional to full-blown programming languages, other examples of formal languages include:

- Simple arithmetic expressions.
- Different ways of specifying dates (such as "Feb. 19, 1999", "February 19, 1999", "le 19 fevrier", "2/19/1999", "today", "next Friday", "last Friday", etc.)
- The implicit language supported by your calculator or VCR. (The "language" consists of the valid sequences of key presses that result in a useful action.)

Not only is all software written in some formal language, many software products also accept as input some kind of formal language.

Formal languages, unlike natural languages, have a well-defined syntax or grammar that is unambiguous and is based on a relatively small number of elementary building blocks.

Version 1.1 (2003-03-11) (Chapter version: 2003-01-24)

An essential feature of languages is that their general from can be succinctly defined with mathematical precision. Once this is done, you can write software to "understand" them and produce some result. Typically, software is written to produce an immediate, understandable result—this is called *interpretation*—or to translate the original language into some other formal language—this is called *translation* or *compilation*.

3.1.1 Some terms

In the following sections, we will use certain terms with fairly precise definitions. Some of the important terms and their definitions follow.

Natural language: What all humans use for ordinary communication from about the age of 2. These languages cannot be understood using the simple methods that work for formal languages[1] (Note: in addition to oral languages like French, sign languages are also natural languages[Pin94].)

Formal language: A sequence of a finite set of symbols that has an unambiguous meaning to a machine.

Grammar: The rules for joining basic symbols into more complex grammatical entities, for joining these complex components into even more complex ones, and so on....(ad infinitum).

Parse: Applying the rules of grammar to an input sentence of a language (natural or formal) and dividing the original sentence into its component parts (and so on recursively). Formal languages can be parsed uniquely; natural languages can often be parsed in multiple ways.

Interpret: Read some sentences from a language and return some result corresponding to its meaning.

Translate (or compile): Translate the language into another language without changing the meaning (i.e. the semantics). Any formal language can be translated into another equally powerful formal language in an unambiguous way; natural languages cannot be translated in this way.

[1] A classic example: What does the sentence "Time flies like an arrow." mean? Find three equally valid meanings. Formal languages, on the other hand, only allow one meaning for any grammatically correct sentence.

When the target language of the translation is designed to be interpreted (yet again) by a machine, we usually call the translator a "compiler".

Meta language: A formal language specifically designed to describe the grammar of any other formal language.

BNF: Backus-Naur Form (BNF) is an example of a meta language.

3.2 Backus-Naur Form

Big fleas have little fleas
That on their bodies bite em
And little fleas have littler fleas
And so ad infinitum

—Jonathon Swift

The grammar for a language describes how syntactical units are defined in terms of other syntactical units or the primitive elements of the language. In English, for example, syntactical units include things like "nouns", "adjectives", "paragraphs", "noun phrases", "sentences", "objects", "subjects', "verbs", etc. The simplest syntactical units in English are words. From a grammatical point of view, you need only know that a word is a "noun", "adjective", "adverb", etc. in order to parse a sentence into its component parts. The elementary components, like words, that cannot be further subdivided are called "terminal symbols".

To summarize:

Terminal symbol: An element of a language that cannot be further subdivided. For example, words in English or numbers in arithmetic expressions are terminal symbols.

Non-terminal symbol: An element that can be sub-divided (parsed) into simpler components. These simpler components may be terminals or themselves non-terminals. For example, a sentence is a non-terminal in English.

3.2.1 Example: Noun phrases in English

We will use the English syntactical unit "noun phrase" to illustrate how BNF is used. A noun phrase is something like "the black cat" or "a hungry dog". From these examples, we can formulate the grammatical rule that a noun phrase is a sequence of an article, an adjective and a noun.

In BNF, we give each non-terminal syntactical unit a unique name and enclose it in angle braces. For example, suitable names for "noun phrase", "noun", "article" and "adjective" could be `<noun_phrase>`, `<noun>`, `<article>` and `<adjective>`. We can then formulate our rule for the syntax or grammar of a noun phrase as:

```
<noun_phrase> ::= <article> <adjective> <noun>
```

When reading a BNF definition, read the symbol `::=` as meaning "is defined as". The above definition reads as "a noun phrase is defined as an article followed by an adjective followed by a noun.

We still have to define what an "article", an "adjective" and a "noun" are. In English, an article is either the word "the" or the word "a". We can describe this in BNF with:

```
<article> ::= 'the'
            | 'a'
```

In this case, the vertical bar symbol (`|`) indicates alternate valid syntax for the unit being defined. Furthermore, each valid alternative in this case a basic English word, i.e. a "terminal symbol". In BNF, terminal symbols are enclosed in single quote marks instead of angle brackets. Thus we would read the above BNF statement as "an article is defined as either the literal word "the" or the literal word "a".

The complete BNF for noun phases is given here (for a limited vocabulary):

```
<noun_phrase> ::= <article> <adjective> <noun>
<article> ::= 'the'
            | 'a'
<adjective> ::= 'hungry'
              | 'black'
              | 'white'
              | 'cute'
<noun> ::= 'cat' | 'dog'
```

We can use this grammar to generate syntactically correct noun phrases or to verify that a given noun phrase is grammatically correct. For example, by substituting different terminals into the rules we can generate noun phrases like "the white dog", "a hungry cat", and so forth. However, we cannot generate "the mangy dog" since "mangy" is not one of allowed adjectives; similarly, "the dog cat" cannot be produced because "dog" is not an adjective.

More importantly, we cannot generate noun phases like "the dog" or "the hungry black cat" because our rule requires exactly one adjective. One way to fix this deficiency would be to define:

```
<noun_phrase> ::= <article> <adjective> <noun>
                | <article> <noun>
                | <article> <adjective> <adjective> <noun>
```

There is, however, a better way. BNF allows parts of a definition to be enclosed in curly braces (`{` and `}`) to indicate that the enclosed element(s) can be repeated zero or more times. Thus, we could define:

```
<noun_phrase> ::= <article> { <adjective> } <noun>
```

We are now allowed to use as many or as few adjectives as we wish.

3.2.2 Example: Arithmetic expressions

Arithmetic expressions that permit adding, subtracting, multiplying, dividing, negating and grouping provide a simple, well-known example of a formal language. We want to show how the grammar of such a formal language can be completely described in the BNF meta-language, and how this description can be used to implement an interpreter or compiler in C (another formal language).

First, we give some examples of valid arithmetic expressions:

```
1
2+1
PI*2*7.2
(((2.0+1.1)/2+5.123)+25*1.5)+123.456)+2)*(6+2))*(2.1 + 5i)
```

Let us first consider a very simple subset of arithmetic expressions—those involving only numbers and addition. Some examples of valid expressions are:

```
3
4.67
2+2
3+4+6
7+8+3+9
12+22+890+13.7+79
```

Some invalid expressions are:

```
4+
++67+
78  9
```

We note that all valid expressions begin with a number; some invalid expressions begin with a number too, so this is not sufficient to identify an expression as valid or not.

For valid expressions, the first number is either followed by nothing or is followed by a '+' operator and another number. In the latter case, the second number is either the end of the expression or is itself followed by a '+' operator and yet another number. In general, we can say that a valid expression starts with a number and is followed by zero or more repetitions of the combination "+ number". Thus we obtain:

```
<expr> ::= <num> { '+' <num> }
```

We can note that from a grammatical point of view, any expression that uses the addition operator will also be grammatically correct if the subtraction operator is substituted for any of the add operators. Using this insight, we can allow for adding and subtracting with the following simple BNF:

```
<expr> ::= <num> { <addop> <num> }
<addop> ::= '+' | '-'
```

We now wish to incorporate multiplication (and division); thus we would like our grammar to accept or generate expressions such as "`3 + 4*7`" or "`3*6*12 - 81*19`".

A simple, but incorrect, solution to this new situation would be:

```
<expr> ::= <num> { <op> <num> }
<op> ::= '+' | '-' | '*' | '/'
```

The problem here is that anything based on this kind of BNF would consider the expressions "1+2*3" and "2*3+1" to be quite different. In particular, a calculator based on the erroneous BNF above would determine that "1+2*3" was "9" and "2*3+1" was "7". The underlying problem is that the add operator has a lower precedence than the multiply operator. Consequently, things joined together with an additive operator have to be treated differently from those joined by a multiplicative operator.

In general, the syntactical units to the left and right of the "+" operator are no longer simple numbers. (For example, "`4*7` is not a simple number, it is a more complex syntactical unit.) We invent a new syntactical unit called "term" which is on either side of an `<addop>` as in the following BNF:

```
<expr> ::= <term> { <addop> <term> }
```

With our current definition, a "term" must be either a simple number or the product of 2 or more numbers. Calling the items that are multiplied together "factors" and allowing for either multiplication or division of the factors, we obtain:

```
<expr> ::= <term> { <addop> <term> }
<term> ::= <factor> { <mulop> <factor> }
<factor> ::= <num>
<addop> ::= '+' | '-'
<mulop> ::= '*' | '/'
```

Now we need to include the left and right parenthesis in or grammar so that expressions like `2.3+5*(10 + 3)` are accepted. Note that if we try to parse this, we end up determining that:

- `5*(10 + 3)` must be a term.
- `5` is a factor and can only multiply other factors; hence `(10 +3)` must parse as a factor. Alas, only `<num>` syntactical units are currently valid factors.
- We note that what is inside the parenthesis—`10 + 3`—is a valid expression. We conclude that a factor can either be a number, as before, or a left parenthesis followed by an expression followed by a right parenthesis.

Version 1.1 (2003-03-11) (Chapter version: 2003-01-24)

The following BNF describes general arithmetic expressions that include everything we want *except* for negation.

```
<expr> ::= <term> { <addop> <term> }
<term> ::= <factor> { <mulop> <factor> }
<factor> ::= <num>
             | '(' <expr> ')'
<addop> ::= '+' | '-'
<mulop> ::= '*' | '/'
```

The BNF is now recursive: `<expr>` is defined as one or more `<term>`s which is defined as one or more `<factor>`s which are either pure numbers or an `<expr>` enclosed in parenthesis.

We can include negation with the following BNF:

```
<expr> ::= <term> { <addop> <term> }
<term> ::= <factor> { <mulop> <factor> }
<factor> ::= <num>
             | '(' <expr> ')'
             | '-' <expr>
<addop> ::= '+' | '-'
<mulop> ::= '*' | '/'
```

The only syntactical unit we have not yet defined is `<num>`. If we are only interested in ordinary numbers such as the `int`s and `float`s supported by a language such as C, we can omit the formal definition and let built-in language facilities (such as the C standard library `scanf` function) deal with the parsing of numbers.

On the other hand, some of the sample arithmetic expressions giving at the beginning of this section do not correspond to C types. The irrational number `PI` is one example and the complex number `2.1 + 5i` is another. If we wanted to support things like complex numbers in our arithmetic expression language, we would have to define BNF for them. (We leave this as an exercise.)

3.2.3 Parse trees

One common way to visualize how a language is parsed is to draw a "parse tree" of valid inputs to the language. The Figures 3.1 and 3.2 give some sample parse trees for arithmetic expressions.

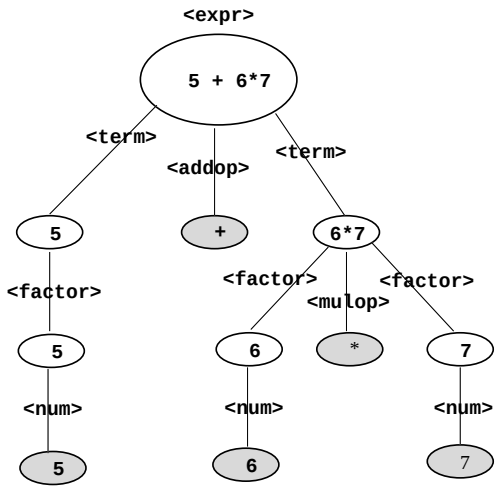


Figure 3.1: Parse tree for `5 + 6*7`

3.3 Implementing a parser/interpreter

The previous sections showed how the BNF description of a grammar is developed. We now show you how to take the BNF description of a formal language and implement a program to parse, interpret or translate sentences of the language.

3.3.1 Overview of Parser Architecture

For every non-terminal syntactical element in the BNF language, we will have a C function with the same name.

For example, if our language is described with:

```
<foobar> ::= <foo> <bar> { <bar> }
<foo> ::= 'a' | 'b'
<bar> ::= 'c' | 'd'
```

then we will have the functions `foobar()`, `foo()` and `bar()` in the C implementation.

Each function will parse the corresponding BNF unit.

The BNF definition of syntactical units allows three general constructs:

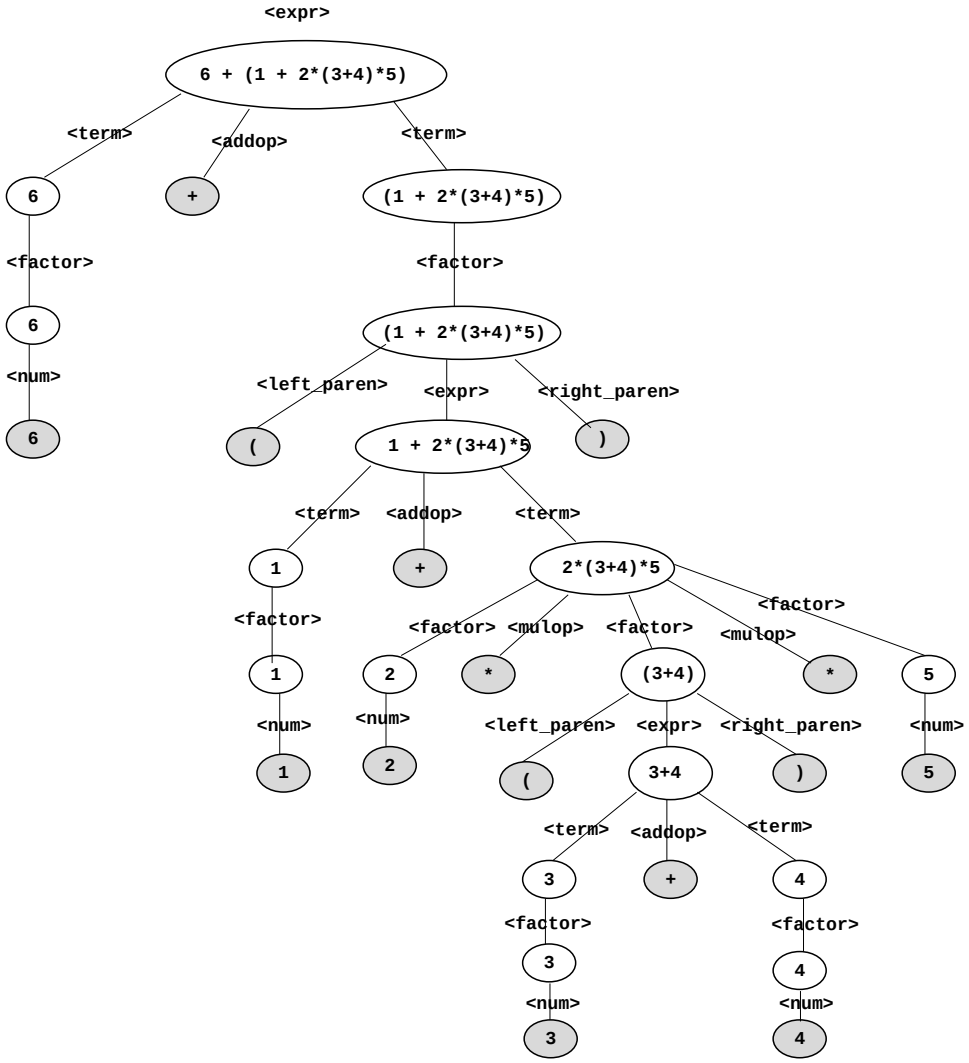


Figure 3.2: Parse tree for `6 + (1 + 2*(3+4)*5)`

Sequence: The input must be parsed as one thing, followed by another, and so on. For example:

```
<thing> ::= <bloop> <bleep> <floop>
```

In this case, the general architecture of the `thing()` function will be:

```
thing()
{
   bloop();
   bleep();
   floop();
}
```

Alternatives: The input must parse as one of two or more alternatives. For example:

```
<colour> ::= <red> | <blue> | <green>
```

The general architecture of the `colour()` function would be:

```
colour()
{
    if ( /* it's a red thing */ )
        red();
    else if ( /* it's a blue thing */ )
        blue();
    else
        green();
```

Note: for simplicity, we assume that the input is always syntactically correct. Thus, in the previous case, if the `<colour>` is not red or blue, then it must be green. We will consider error detection later; the assumption of grammatically correct input simplifies our job for the moment.

Repetition: The input must parse as a repetition of zero or more things. For example:

```
<many> ::= { <one> }
```

In this case, the architecture of the `many()` function would be:

```
many()
{
     while ( /* it's a ``one'' */ )
        one();
}
```

Of course, the basic constructs can be combined. For example, given the BNF for noun phrases described earlier:

```
<noun_phrase> ::= <article> { <adjective> } <noun>
<article> ::= 'the'
              | 'a'
<adjective> ::= 'hungry'
               | 'black'
               | 'white'
               | 'cute'
<noun> ::= 'cat' | 'dog'
```

We can describe the overall architecture of a parser as:

```
nonu_phrase()
{
     article();
     while ( /* next thing is an adjective */ ) {
        adjective();
     }
     noun();
}

article()
{
   if ( /* next thing is "the" */ )
        /* process "the" */
   else
        /* process "a" */
```

```
}

adjective()
{
   if ( /* next thing is "hungry" */ )
        /* process "hungry" */
   else if ( /* it's "black" */
        /* process "black" */
   else  /* and so on for other valid adjectives */
}

noun()
{
   if ( /* next thing is "cat" */ )
        /* process "cat" */
   else
        /* process "dog" */
}
```

3.3.2 Moving on—Tokens and Actions

The overall architecture of a parser program can be clearly defined by knowing the BNF. We do not yet have a working program, however, as we have not shown how one gets the basic elements from the input sentence and how these are processed.

We use the word "action" to describe how an element is processed. All programs that parse a formal language will have the same general architectural form described above[2]. However, programs that translate, compile or interpret their input will perform quite different actions as they parse their input.

For example, suppose we want to write a program to parse arithmetic expressions described by the BNF we developed previously. An interpreter would parse the expression and print out a number corresponding to its value;

[2]This is only true for *recursive descent parsers*—the formal name for the kind of parsers we discuss here. There are, however, formal languages that cannot be easily parsed with this technique. You will have to take a more advanced course on "Compilers and Translators" to appreciate this distinction.

a translator program could output the expression in *Reverse Polish Notation* (RPN).

When an interpreter program encountered a `<num>` element, its action would be to transform the textual encoding of the number into some machine representation of the number itself[3]. On the other hand, an RPN translator program may simply output the character as its action.

The discussion of parsers thus far has also omitted the question of how and when the input language is read.

We consider the raw input to be a stream of characters. We do not care where the stream is coming from—it may be coming from `<stdin>`, from an array of characters, or from a URL on the Internet. Conceptually, however, we consider the input to be a stream of *tokens*, not mere characters. We define a token as any of the basic terminal symbols of the language. For example, the tokens for the "noun phrase language" described earlier would be the words "a", "the", "cat", "dog", "black", "white", "hungry" and "cute." The arithmetic expression language has the tokens "`+`", "`-`", "`*`", "`/`" "`(`", "`)`" and "numbers". In the case of arithmetic expressions, it is also desirable to allow arbitrary numbers (including zero) or whitespace characters between the tokens; the noun phrase language, of course, needs at least one space between the words, but more than one space should be acceptable as well.

Author's note: More discussion about the similarities and differences between "tokens" and "terminal symbols" would be useful...

In general, the transformation of the input stream of raw characters to a stream of tokens fed to the parser is a separate phase. This phase is called "lexical analysis", "tokenizing" or "scanning". In the simple languages considered here we will encapsulate this phase into a single function—`getNextToken(void)`—which will read the raw characters from the input stream, return the next token and perform basic housekeeping operations such as ignoring unnecessary whitespace.

There still remain two questions: what does `getNextToken()` return and when should it be called?

The return data type is the simplest question; we will simply have `getNextToken()` return the "token type". The nature of the token type will vary depending on the formal language. Sometimes it will be a string (e.g. in a noun phrase tokenizer), sometimes it will be an `int` and sometimes will will be a `struct` or even something else. The precise nature of the data type will be encapsulated with a `typedef` such as:

[3]For example, a number represented as a single character might be processed with: `n = ch - '0';`

```
typedef int token_t;
```

or

```
typedef char * token_t;
```

or

```
typedef struct { .... }  token_t;
```

The final unresolved issue involves the interaction between the tokenizer and the parser: when does the parser ask for the next token?

To see why this is not a completely trivial question, consider the general architecture of the repetition parser:

```
many()
{
     while ( /* it's a ``one'' */ )
        one();
}
```

To escape the "while loop", we have to encounter a syntactical unit that does not parse as a `<one>`. This necessarily implies that we must gather one or more tokens that extend beyond the end of the `<many>` unit.

There are at least two ways to resolve this problem:

1. Allow some mechanism to "unget" tokens—i.e. to "push them back" onto the input stream of tokens. The decision of what to push back and when to do it can be localized in the individual parsing functions; some functions need never push things back while others do. Every function is written as if the token returned by `getNextToken()` is the next unexamined token.

   This is arguably the most elegant solution as it avoids any global variables and unexpected interaction between parsing functions. However, its implementation requires some sophistication in C programming and modularization skills that may be a barrier to students with scant experience with these techniques.

2. Use a global variable—`token_t token`—that by convention you always set whenever you get another token. (i.e. You use the pattern `token = getNextToken()`.) This technique inevitably means that there is some interaction between the parsing functions that the programmer using them must be aware of. Consequently, it is essential that the public documentation of each parsing function clearly state if the global variable `token` has been successfully processed or not—i.e. does the calling function need to get the next token to continue parsing or is it already available in the global variable.

   We adopt this method.

3.3.3 Example: A Noun Phrase Word Counter

With these preliminaries, we are now able to write a parser for a language. The precise task of the parser will be to implement an interpreter: specifically, the program will read an alleged noun phrase from `<stdin>` and print "`Good: <n> words`" when the input phrase is a grammatically correct noun phrase and where `<n>` is a count of the total number of words in the phrase. If the input phrase is ungrammatical, the message "`Bad`" will be printed.

Note that this apparently trivial program is quite different form a simple word count program (like the UNIX `wc -w` command); the ordinary `wc -w` will happily report that the phrase "dog white the" has 3 words in it, but our parser program will say that this phrase is `Bad`. Of course, when the phrase is grammatically correct, both programs will give the same numerical answer.

We will assume that a tokenizer exists with the public interface previously described. Specifically:

```
typedef char * token_t;
token_t token;
token_t getNextToken(void);
```

and the public documentation for `getNextToken()` is:

```
/** getNextToken() reads the next word from stdin and
 * returns a pointer to it.  Whitespace (spaces and tabs)
 * are ignored.  If the end of file is encountered, an
 * empty string is returned.  The end-of-line indicator
 * is returned as the string "\n".
```

```
  *
  * @return a string or "" or "\n" as described above.
  */
```

We now need to define what each of the language specific functions (`noun_phrase`, `article`, `adjective` and `noun`) will return and how they will manipulate the global `token` variable.

Recalling the importance of clearly specifying when tokens are retrieved, the first step in writing C functions for each of the syntactical elements is to write clear public documentation. For example, the public documentation to the `noun_phrase()` function is:

```
/** noun_phrase() parses a noun phrase defined by the BNF:
 * <pre>
 * <noun_phrase> ::= <article> { <adjective> } <noun>
 * </pre>
 *
 * ENTRY CONDITIONS: The next unprocessed token must be
 *                   already available in the token
 *                   global variable.
 * EXIT CONDITIONS:  The next unprocessed token will be placed
 *                   in the token global variable.
 * @return The number of words in the noun phrase
 *         (must be at least 2) or a negative number
 *         if a parse error is detected.
 */
```

With this documentation (and similar documentation indicating the responsibilities for getting the next token for other functions), we can now easily write the `noun_phrase()` function—assuming that the input is grammatically correct—as follows:

```
int noun_phrase(void)
{
  int nWords;
  nWords = article();
  while(adjective())
    nWords++;
  noun();
  return nWords+1;
```

Version 1.1 (2003-03-11) (Chapter version: 2003-01-24)

If you look at the complete source code, you will see that the implementation differs from this simplified version. In particular, grammatically correct input is not assumed. When the input is ungrammatical, a negative value is returned.

The assumption that the input is grammatically correct simplifies our life considerably. For example, the public documentation for the `article()` function states:

```
/** article() parses an article as defined by:
 * <pre>
 *<article> ::= 'the' | 'a'
 * </pre>
 * ENTRY CONDITIONS: The next unprocessed token must be already
 *                   available in the token global variable.
 * EXIT CONDITIONS:  The next unprocessed token will be placed
 *                   in the token global variable.
 * @return 1 if the token is an article; otherwise 0.
 */
```

Under the assumption of grammatically correct input, the implementation of the `article()` can be simplified to simply getting the next token and unconditionally returning 1 as follows:

```
int article(void)
{
    token = getNextToken();
    return 1;
}
```

Indeed, this kind of "stub function" was used during the development of the program.

The `main()` function

The discussion so far has implicitly assumed that the parser functions are simply given a sequence of tokens from the lexical analyzer (i.e. the `getNextToken()` function) that parse as a `<noun_phrase>`. However, we also need a driver—a "main" function—to get things going. Furthermore, the specifications for the program mandate that it can handle zero or more noun phrases, each on a line by itself, and that empty lines be ignored.

We can express these constraints for our driver program with the following (informal) BNF:

```
<line> ::= '\n' | <noun_phrase> \n
<a_bunch_of_lines> ::= { <line> } EOF
```

Frankly, however, I did not go through this BNF stage; I simply wrote the `main()` function using common C idioms as follows:

```
int main()
{
  int nWords;
     /* while NOT end-of-file */
  while(strcmp(token = getNextToken(), "")) {
    if (!strcmp(token, "\n")) /* Ignore empty lines */
      continue;
    if((nWords = noun_phrase()) > 1)
      printf("Good: %d words\n", nWords);
    else {
      printf("Bad\n");
      /* Skip rest of line */
      while(strcmp(token, "\n") != 0)
        token = getNextToken();
    }
  }
  exit(0);
}
```

3.3.4 Example: A Simple Calculator

The BNF for simple arithmetic expressions can also be converted into a working C program following the same general principles we applied for the noun phrase parser.

For example, the BNF:

```
<expr> ::= <term> { <addop> <term> }
<term> ::= <factor> { <mulop> <factor> }
<factor> ::= <num>
             | '(' <expr> ')'
```

```
<addop> ::= '+' | '-'
<mulop> ::= '*' | '/'
```

has the overall architecture:

```
expr()
{
    term();
    while (/* next thing is <addop> */)
       term();
}

term()
{
    factor();
    while (/* next thing is <mulop> */)
       factor();
}

factor()
{
   if (/* next thing is a number */)
       /* process number */
   else {
      /* next thing must be '(', read it */
      expr();
      /* next thing must be ')', read it */
}
```

Unlike the noun phrase parser, this architecture is recursive. This is not *direct recursion*—the type we discussed in Chapter 2 where a function directly calls itself—rather it is *indirect recursion*. In this case there is a circular chain of function calls that can result in the same functions being invoked again before the first invocation has returned. Here, `expr` calls `term` which calls `factor` which may again call `expr`...

To implement a calculator based on this general parser architecture, we need to define the actions and tokens.

We will use a very simple token type: the tokens will simply be single characters (the only valid ones are "`0123456789-+*/()`"[4]) and the tokenizer will skip whitespace. The implementation is:

```
typedef int token_t;

token_t getNextToken(void)
{
  int ch;
  /* Skip spaces and tabs */
  while (((ch = getchar()) == ' ') || (ch == '\t'))
    ;
  return ch;
}
```

The actions for the non-terminals `<expr>`, `<term>` and `<factor>` will be to return the numerical value of the corresponding syntactical unit.

For example, when we implement `expr`, the action associated with parsing the initial `<term>` should be to remember its return value (which we call `value`.) If there is nothing more to the expression (i.e. if the `<term>` is not followed by any `<addop> <term>` pairs, we simply return this `value`.

If, on the other hand, the initial term is followed by an `<addop> <term>` pair, we should parse the second term, get its value and either add it or subtract it from the first term depending on whether the `<addop>` was a '+' or a '–'.

For example, if the expression to parse is "$3+4$", the first term evaluates to "3" and is followed by a `<addop> <term>` pair. We remember that we have to perform an addition, then parse the `<term>` following the '+' (i.e. the "4") returning the value 4 which we add to the "3" to obtain the final value for the `<expr>`. Of course, had the first term been followed by a $-$ instead of a $+$, we would have subtracted the value of the second term from the first.

The C code to implement the interpreter for an `<expr>` is:

```
int expr(void)
{
```

[4]The end-of-file indicator—EOF—is also a valid token. Because of this, the tokens must be of type `int`, not `char`.

```
  int value,  valueRight;
  token_t opToken;

  value = term();
  while((opToken = token) == '+' || opToken == '-') {
    token = getNextToken();
    valueRight = term();
    if(opToken == '+')
      value = value + valueRight;
    else
      value = value - valueRight;
  }
  return value;
}
```

The code to interpret a `<term>` is similar to that for an `<expr>` except that we multiply or divide in the `while` loop instead of adding or subtracting.

The code to interpret a factor is quite different, however, since it involves alternatives and is where numbers themselves are interpreted. Recall that tokens are single characters and, hence, the only numbers that can be entered into the interpreter are the ten single digit numbers encoded by the characters '0'—'9'. These are *encodings* for numbers (usually in ASCII), not numbers themselves. To convert the encoding for a single-digit number to the actual value of the number, we subtract the encoding for the character '0' from it. (We subtract '0' rather than 0x30 or 48 because it is better written code and is not dependent on the ASCII encoding scheme.)

The C code to interpret a `<factor>` is:

```
int factor(void)
{
  int value;

  if (isdigit(token)) {
    value = token - '0';
    token = getNextToken();
    return value;
  } else {
    assert(token == '(');
```

```
    token = getNextToken();
    value = expr();
    assert(token == ')');
    token = getNextToken();
  }
  return value;
}
```

Finally, the main driver routine is:

```
int main()
{
  int value;
  while ((token = getNextToken()) != EOF) {
    if (token == '\n')
      continue;
    value = expr();
    printf("%d\n", value);
  }
  exit(0);
}
```

(The complete source code is given in Appendix E or in the file `src/parsing/calc.c`.)

3.4 Problems

3.1 Does the software on your calculator implement an interpreter or a translator?

3.2 Is the process of converting C source code to object code for a specific machine best described as "interpretation", "translation" or "compilation"?

3.3 The BNF grammar for C in Appendix A of Kernighan and Ritchie defines an "if statement" as follows:

```
<if_statement> ::= 'if' '(' <expression> ')' <statement>
```

Using this as a model, define the syntax for a "while statement".

Version 1.1 (2003-03-11) (Chapter version: 2003-01-24)

3.4 Add the exponentiation operator (denoted with a `^`) to the arithmetic expression BNF. For example, valid expressions would now include things like `2^(1+3)*5` which would evaluate to `80`.

3.5 Describe the BNF for complex numbers. Include two alternate input formats: one for rectangular notation and the other for polar notation. You may assume that ordinary floating point numbers have already been defined as `<num>` syntactical units.

3.6 Describe the BNF for unsigned floating point numbers as understood by C's `scanf` function in terms of a sequence of characters. (Ignore the restrictions to size or precision.)

3.7 Assume that C's "if" and "while" statements have already been defined as outlined previously.

1. Define the BNF syntax for an "assignment statement". Assume that the units `<id>` and `<expression>` which describe respectively the syntax of valid variable names and arithmetic expressions have already been defined.

2. What else do you have to add to the grammar so that C language constructs such as:

```
if(...)
   foo = bar + 2;
```

   and

```
if(...) {
   foo = bar + 2;
   while (...) {
      while(...)
         n = n - 1;
      p = p - 2;
   }
}
```

   are valid?

3.8 Name at least 15 tokens in the C programming language. The tokens can be keywords, operators, or separators.

3.9 Which programming language do you know with the least number of tokens? ....the most number of tokens?

3.10 The `nounPhraseWordCounter` program correctly identifies "the black cat" as a valid noun phrase consisting of 3 words, but it produces a parse error message for the phrase "The black cat." Why? Which function should you modify to fix this problem for the general case of ignoring the case of the first letter in a word? Modify the function.

3.11 Suppose that it is determined that 90% of valid noun phrases begin with the word "the" (10% begin with "a"). Suppose also that only 20% of all input sentences have a valid initial word.

1. Is there anything you can do to the source code so that valid sentences are detected more quickly?

2. Is there anything you can do to the source code so that invalid sentences are detected more quickly?

3.12 Suppose that all sentences are valid and that the frequency of adjectives is "white": 40%, "cute": 30%, "hungry": 20% and "black": 10%. Can you make the program more efficient on average knowing these probabilities. Explain.

3.13 Identify some bugs in the nounPhraseWordCounter program. (I am aware of at least two...)

3.14 How would you modify `calc.c` so that floating point numbers (like 1.23E-5) could be used instead of single digit integers. Implement it in C.

3.15 Draw parse trees for each of the following arithmetic expressions:

1. $---(1+2)$
2. $1+2*3$
3. $(1+2)*3$
4. $1+2*(3--(4+5))$

3.16 Modify the implementation of the simple calculator to include the unary minus operator.

3.17 Define BNF so that sentences like "The hungry white dog chased the cute black cat." could be recognized. (Hint: you will need to include a few examples of verbs.)

Version 1.1 (2003-03-11) (Chapter version: 2003-01-24)

Chapter 4

Complexity

We saw in Chapter 1 that algorithms can be classified as *linear*, *quadratic*, *logarithmic*, etc. depending on the time they take as a function of a number denoting the size of the problem. We showed, for example, that the selection sort algorithm was quadratic and that merge sort was of $n \lg n$ complexity. We also played "fast and loose" with our analysis, using simplifications such as "ignore the time for this operation", assume another operation takes precisely 1 time unit, did some "hand waving" to obtain manageable equations which we then solved to prove our statements.

In this chapter, we will put these ideas on a firm mathematical foundation, introduce some new notation, examine some ways to solve recurrences and give several examples.

4.1 Basic Concepts

The basic concepts behind *complexity characterization* of algorithms can be illustrated with reference to the following virtual algorithm:

Example Algorithm

An undefined virtual algorithm

Step 1: Do this step in time k_1. (*The step is only done once.*

Step 2: This step takes k_2 time. (*This step is done n times.*)

Version 1.1 (2003-03-11) (Chapter version: 2003-03-11)

Step 3: This step takes k_3 time. (*This step is done* n^2 *times.*)

Step 4: This step takes k_4 time. (*This step is done* n^3 *times.*)

No details are specified about what the algorithm does; nonetheless, we can conclude some general characteristics about it:

$T(n) = k_4n^3 + k_3n^2 + k_2n + k_1$

where: $T(n)$ is the time to perform the algorithm for problem size n.

Since $T(n)$ is a cubic equation, we summarize the time complexity of the algorithm by saying that it is cubic. In this chapter, we will examine various mathematical notations so that this English description is expressed mathematically as $T(n) = \Theta(n^3)$.

4.1.1 Average-, Worst- or Best-case Analysis

When analyzing the number of times each step is performed in an algorithm for problem size n, different results will often be obtained depending on whether the best-, worst- or average-case is considered.

The most conservative type of analysis is the *worst-case* situation. We can then *guarantee* the performance no matter what the input is.

Average-case performance is also useful; indeed, it is sometimes even more useful than worst-case analysis. However, average-case performance metrics do not provide the absolute guarantee given by worst-case analysis. Average-case analysis can also be more difficult than worst-case analysis.

In short, algorithms are usually compared for the worst case. Generally speaking, an algorithm that performs with logarithmic complexity in the worst case is preferable to one that is linear. In this book, we will almost always be concerned with worst-case analysis.

But all rules have exceptions. For example, in the worst case the merge-sort algorithm has complexity $n \log n$ while the more widely used Quicksort algorithm has n^2 worst-case complexity. Quicksort, however, has *average* $n \log n$ complexity and the worst-case situation is highly unlikely (especially if the input is randomized.) Quicksort also has lower memory requirements and almost always is faster than merge-sort.

Besides worst- and average-case metrics, best-case analysis can also be done. However, there is scant justification for best-case analysis, except when it is compared with worst- or average-case analysis. Furthermore, it is easy

to abuse best-case analysis by modifying an algorithm to treat special "best-case" inputs in some unusual optimized way.[1] Despite these caveats, best-case analysis (without fudging the algorithm) is sometimes a useful technique to flesh-out some special characteristics of algorithm performance and to compare them with what is encountered in the average and worst cases.

4.1.2 Time and Space complexity

In almost all cases, we will analyze the time performance of algorithms whose size can be characterized with a single parameter, n.

However, there are occasions when this does not apply.

For example, we can verify that a simple algorithm to add up n numbers can be done in a time proportional to n. This is only true, however, if the computer performing the algorithm can add two numbers in some constant time. For modern computers, this will be true for integers less than about 4 billion.

Suppose, however, that the integers may be arbitrarily large; perhaps some of the numbers are hundreds or thousands of digits long. In this case, the computer hardware cannot add the numbers directly in a single clock cycle. Rather, software (another algorithm) is required to perform arithmetic on such huge numbers.

It can be shown that the time required to add huge numbers is proportional to the number of digits, p, involved. The number of additions is proportional to n and the time per addition is proportional to p. Consequently, the total time is a function of two size parameters, n and p, and the time required for the algorithm would be proportional to np.

Finally, although algorithm complexity usually means a measure of the time required to do the algorithm, it is sometimes useful to calculate the amount of memory required to solve a problem of size n. This is called the *space-complexity* of the algorithm. In this book, any discussion of space-complexity will explicitly use this term; otherwise, the term *complexity* will implicitly mean *time-complexity*.

[1] There have been reports that some unscrupulous software companies whose products are often compared using standardized benchmarks have modified their algorithms to detect well-known benchmark inputs and generate the desired result in a specialized way.

4.2 $O()$ notation: asymptotic upper bound

Suppose that $T(n)$ is an equation for the running time of some algorithm where n is the size of the problem. We say that $T(n) = O(g(n))$ to mean that the time required to complete the algorithm is no worse than some *multiple* of $g(n)$ for *sufficiently large* problems (as measured by n.)

Example

We can say that $T(n) = 2n^2 + 5n = O(n^2)$ because:

$$2n^2 + 5n < 10 \times n^2$$

for all $n > 1000$.

In this example, we have chosen to multiply $g(n) = n^2$ by 10 and define "sufficiently large" as greater than 1000.

4.2.1 Formal definition of $O()$

The formal definition of $f(n) = O(g(n))$ is the set of functions, $g(n)$ such that

$$f(n) < kg(n), \forall n > n_0$$

where k and n_0 are constants. Note that the specific values of the constants k and n_0 depend on the $g(n)$ function chosen from the set. This formal definition can then be used to prove that a function has some $O()$ property.

Example Show that $3n^2 + n$ is $O(n^2)$.

Choose $g(n) = 4n^2$ and find a value n_0 such that $f(n) > g(n)$ for all $n > n_0$. We have:

$$\begin{aligned} 4n^2 &> 3n^2 + n \\ 4n &> 3n + 1 \\ 4n - 3n &> 1 \\ n &> 1 \end{aligned}$$

Clearly, $n_0 = 1$. If you were to choose $k = 2$ (i.e. $g(n) = 2n^2$) you would be unable to find any value for n_0. (If you are unconvinced of this, try it and see why.)

Example Show that $3n^2 + n$ is $O(n^3)$.

Choose $g(n) = 2n^3$ and find a value n_0 such that $f(n) > g(n)$ for all $n > n_0$. We have:

$$\begin{aligned} 2n^3 &> 3n^2 + n \\ 2n^2 &> 3n + 1 \\ 2n^2 - 3n &> 1 \end{aligned}$$

Choose $n_0 = 2$, so we have $n > n_0 = 2 \Longrightarrow n^2 > 2n$. Substituting in the above inequalities, we obtain:

$$\begin{aligned} 2 \times 2n - 3n &> 1 \\ n &> 1 \end{aligned}$$

which is obviously true for $n > n_0 = 2$

4.2.2 Remarks about $O()$ notation

The examples illustrate some curious characteristics of $O()$ notation. For example, given $T(n) = 2n^3 + 5n^2 + 10$, all of the following statements are valid:

- $T(n) = O(73n^4)$
- $T(n) = O(2n^3)$
- $T(n) = O(2n^3 + 5n^2)$
- $T(n) = O(n^3)$

Although these are all mathematically correct, it is usually silly to use anything but the simplest $O(g(n))$ expression where $g(n)$ is the slowest growing possible function. In this case, the best characterization of $T(n)$ is $O(n^3)$.

It may also seem curious that we can say (quite correctly) that $T(n) = O(n^3)$ and $T(n) = O(n^8)$. Both are valid because $O()$ notation establishes an upper bound on the growth of the function; clearly, it a function is bounded such that it is smaller than kn^3 for some constant, k, then it will also be bound by some function $k_1 n^8$.

The reader may well ask why anyone in their right mind would ever characterize $2n^3 + 5n^2$ as $O(n^8)$. Certainly, if you wanted to increase the popularity of an algorithm, you would be well-advised to make the valid claim that it is $O(n^3)$ instead of saying it is $O(n^8)$. (The situation is similar to trying to promote someone who is 7'1" tall for a basketball team by saying either "He's more than seven feet" vs. the equally true, "He's more than three feet high.")

Why then, you may ask, would anyone ever not use the most attractive possible $O()$ notation? Only rarely... One situation that does occur is when the theoretical analysis of an algorithm's complexity is mathematically very difficult. Sometimes, the analysis can be simplified by using assumptions that guarantee performance that is no better (and may be worse) but that make the mathematics manageable. In the end, you may be able to prove that the algorithm is, say, $O(n^3)$ but there remains the possibility that its behavior is even better.

4.2.3 Tips for determining $O()$ complexity

The $O()$ complexity of many functions can be determined by inspection in many cases. (Furthermore, the tips shown here also apply to $\Omega()$ and $\Theta()$ notations which we will be examined shortly.)

First, the following general properties of $O()$ notation can be proved:

1. If $g(n) = O(G(n))$ and $f(n) = O(F(n))$, then:

$$f(n)+g(n) = O(F(n))+O(G(n)) = O(F(n)+G(n)) = O(\max(F(n), G(n)))$$

2. If $g(n) = O(kG(n))$ (where k is a constant), then $g(n) = O(G(n))$.

Using these facts, it is simple to show that

$$\begin{aligned} T(n) &= 2n^3 + 5n^2 + 10 \\ &= O(2n^3 + 5n^2 + 10) \end{aligned}$$

$$\begin{aligned} &= O(2n^3) + O(5n^2) + O(10) \\ &= O(n^3) + O(n^2) + O(1) \\ &= O(n^3) \end{aligned}$$

Usually these formal steps are skipped: simply identify the fastest growing term, ignore constant multipliers and you get $O()$ notation.

Indeed, for the case of a function that is the sum of terms that are "simple" (involving, for example, only polynomials, logarithms and exponentials), this straightforward method also yields $\Omega()$ and $\Theta()$ complexity[2]

4.3 $\Omega()$ and $\Theta()$ notations

Characterizing a function with $O()$ notation establishes a *loose upper asymptotic bound* on its growth for large n.

Two other notations are also widely used:

$\Omega()$: establishes a *loose lower asymptotic bound.*

$\Theta()$: establishes a *tight asymptotic bound.*

$\Omega()$ notation is the opposite of $O()$ notation: when we say a function, $f(n)$ has $\Omega(g(n))$ complexity, we mean that $f(n)$ grows at least as fast (and possibly faster) than $g(n)$ for large n.

The formal definition of $f(n) = \Omega(g(n))$ is the set of functions, $g(n)$ such that

$$f(n) > kg(n), \forall n > n_0$$

where k and n_0 are constants.

As stated earlier, the simple method for determining $O()$ complexity is also true for $\Omega()$ notation. Thus:

$$\begin{aligned} T(n) &= 2n^3 + 5n^2 + 10 \\ &= \Omega(2n^3 + 5n^2 + 10) \\ &= \Omega(2n^3) + \Omega(5n^2) + \Omega(10) \\ &= \Omega(n^3) + \Omega(n^2) + \Omega(1) \\ &= \Omega(n^3) \end{aligned}$$

[2]We examine $\Theta()$ and $\Omega()$ complexity shortly.

On seeing this, the reader is excused if they wonder "What is all this silliness!"

Before answering, note also that the general rule we used previously, i.e.:

$$f(n)+g(n) = O(F(n))+O(G(n)) = O(F(n)+G(n)) = O(\max(F(n), G(n)))$$

is also valid for $\Omega()$-notation:

$$f(n)+g(n) = \Omega(F(n))+\Omega(G(n)) = \Omega(F(n)+G(n)) = \Omega(\max(F(n), G(n)))$$

But, in addition, the following rule is also valid:

$$f(n) + g(n) = \Omega(F(n)) + \Omega(G(n)) = \Omega(\min(F(n), G(n)))$$

That is, we can simplify $\Omega()$ expressions by ignoring all but the *slowest* growing term.

Indeed, recall that in the case of $O()$ notation when $f(n) = O(g(n))$ is established, then it is also true that $f(n) = O(g'(n))$ where $g'(n)$ is any function that grows as fast or faster then $g(n)$. The analogous statement for $\Omega()$ notation is that once $f(n) = \Omega(g(n))$ is established, then it is also true that $f(n) = \Omega(g''(n))$ where $g''(n)$ is any function that grows as slow or *slower* then $g(n)$.

One curious consequence of this is that essentially all functions (except for $f(n) = 0$!) is $\Omega(1)$.

The reader may still wonder what use this notation has; bear with me a little bit more and some useful applications will be given.

4.3.1 $\Theta()$ notation

$\Theta()$ notation is the most useful way to characterize the growth rate of a function because it set *tight bounds* rather than the loose upper (or lower) bounds given by $O()$ (or $\Omega()$.) Basically, it provides a function that when multiplied by one constant gives the lower bound and when another multiplier is used (on the *same* function, an upper bound is set. In short, $\Theta()$ combines the features of $\Omega()$ and $O()$.

The formal definition of $\Theta(f(n))$ is the set of functions, $g(n)$ and three constants, k_1, k_2 and n_0, such that

$$k_1 g(n) < f(n) < k_2 g(n), \forall n > n_0$$

Once again, when we need to determine the $\Theta()$ complexity of a function that is the sum of "simple" terms, we can simply choose the fastest growing term and set any constant multipliers to unity to obtain the $\Theta()$ complexity.

4.4 Remarks on $\Theta()$, $O()$ and $\Omega()$ notations

4.4.1 Basic properties

The following basic properties apply to asymptotic notations.

Transitivity: All notations are transitive. For example:

$$\text{if } f(n) = O(g(n)) \text{ and } g(n) = O(h(n)) \Longrightarrow O(h(n))$$

Reflexivity: All notations are reflexive. Thus:

$$f(n) = O(f(n))$$
$$f(n) = \Omega(f(n))$$
$$f(n) = \Theta(f(n))$$

Symmetry: Only $\Theta()$ notation is symmetric, i.e.:

$$f(n) = \Theta(g(n)) \text{ if and only if } g(n) = \Theta(f(n))$$

Ordering analogies: It is often useful to think of the relationship between the asymptotic notations as analogous to the ordering relationships (less than, equals, greater than) between ordinary numbers. For example:

$$O() \text{ "means" } \preceq \text{ or } f(n) = O(g(n)) \Leftrightarrow f \preceq g$$
$$\Omega() \text{ "means" } \succeq \text{ or } f(n) = \Omega(g(n)) \Leftrightarrow f \succeq g$$
$$\Theta() \text{ "means" } \approx \text{ or } f(n) = \Theta(g(n)) \Leftrightarrow f \approx g$$

Transpose symmetry: The following transpose symmetry relationships apply (and are easily seen by using the above analogies.)

$$f(n) = O(g(n)) \Leftrightarrow g(n) = \Omega(f(n))$$
$$f(n) = \Theta(g(n)) \Leftrightarrow f(n) = \Omega(g(n)) \text{ and } f(n) = O(g(n))$$

4.4.2 When to use $O()$, $\Omega()$ and $\Theta()$

In general, the preferred analysis is *worst-case* $\Theta()$. However, there are situations where this is difficult or when average-case analysis is preferable.

Example: $n!$

Consider, for example, the problem of determining $\Theta(n!)$. Without additional knowledge of mathematics (such as Stirling's approximation), this can be daunting. However, finding simple expressions for $O(n!)$ and $\Omega(n!)$ are quite easy.

First, to obtain $O(n!)$, note that:

$$\begin{aligned} n! &= n \times (n-1) \times (n-2) \cdots 2 \times 1 \\ &< \underbrace{n \times n \times \cdots \times n}_{n\text{times}} \\ &= n^n \end{aligned}$$

Clearly, then:

$$n! = O(n^n)$$

Similarly, we can obtain $\Omega(n!)$ as follows:

$$\begin{aligned} n! &= n \times (n-1) \times (n-2) \cdots 2 \times 1 \\ &> \underbrace{2 \times 2 \times \cdots \times 2}_{n\text{times}} \\ &= 2^n \end{aligned}$$

Hence,

$$n! = \Omega(2^n)$$

(Indeed, $n! = \Omega(k^n)$ for any k where $n > 2k$.)
However, it can also be shown that:

$$\begin{aligned} n! &= O(n^n) \text{ but } n! \neq \Omega(n^n) \\ n! &= \Omega(2^n) \text{ but } n! \neq O(2^n) \end{aligned}$$

Hence, this analysis does not provide a $\Theta()$ expression for $n!$ although it does tell us that it does grow faster than any exponential (k^n) but not as fast as n^n.

One way to obtain a tight asymptotic bound is to use *Stirling's approximation*:

$$n! = \sqrt{2n\pi}\left(\frac{n}{e}\right)^n (1 + \Theta(1/n))$$

or,

$$\sqrt{2n\pi}\left(\frac{n}{e}\right)^n < n! < \sqrt{2n\pi}\left(\frac{n}{e}\right)^{n+\frac{1}{12n}}$$

Using the first version, we can obtain:

$$n! = \Theta\left(\sqrt{n}\left(\frac{n}{e}\right)^n\right)$$

Example: $\log n!$

Unlike the factorial function itself, it is possible to obtain a tight asymptotic bound for the logarithm of a factorial—$\log n!$—without recourse to Stirling's approximation as follows:

$$\begin{aligned} \log(n/2)^{n/2} < \log n! < \log n^n \\ (n/2)(\log n - \log 2) < \log n! < n \log n \\ kn \log n < \log n! < n \log n \end{aligned}$$

Version 1.1 (2003-03-11) (Chapter version: 2003-03-11)

Hence:

$$\log n! = \Theta(n \log n)$$

This is a significant result of practical importance. In particular, this result can be used to prove that no sorting algorithm that relies on pairwise comparisons of elements can have better than $\Theta(n \log n)$ complexity.

Any sort algorithm for n objects must choose which of the $n!$ possible permutations of the objects is the one in which all the elements are in sorted order. An algorithm based on pairwise comparisons must ultimately descend a decision tree until it encounters the correct permutation. Because $\log n! = \Theta(n \log n)$ the best height for such a decision tree is also $\Theta(n \log n)$ and since the "best height" corresponds to the "best algorithm", no sort algorithm of this type can do better than $\Theta(n \log n)$.

4.5 Analysis of non-recursive algorithms

The simplest kind of algorithm to analyze contains no loops and only elementary steps. (Conditional goto steps are allowed, however, so long as they cannot result in a loop.) In this case, the running time is always $\Theta(1)$ (and $O(1)$ and $\Omega(1)$).

This is easy to prove in general. If there are no loops, each step S_i is performed at most once. Since each step is elementary, we can assign an upper bound on the time to perform it, T_i. Consequently, we have:

$$T(n) = \sum_{\text{each step}} T_i = \text{some constant}$$

This represents the *absolute worst case*. Suppose, for example, that Step 1 is always performed (in time T_1) and then either Step 2 or Step 3 is performed. The *real* worst case is then:

$$T(n) = T_1 + \max(T_2, T_3)$$

If the probabilities of doing Steps 2 and 3 were p_2 and p_3 respectively, the average time would be:

$$T(n) = T_1 + p_2 T_2 + p_3 T_3$$

The 8-step algorithm for calculating the date for Easter (Problem 1.11) is an example of an algorithm of complexity $\Theta(1)$. When the algorithm is

written in C and compiled to machine language for a SPARC processor, there are about 130 machine instructions used to implement the algorithm. Assuming that each machine instruction takes about 5 nanoseconds to execute, the algorithm should be executed in a little less than 1 microsecond. When the execution time was measured, it was 0.87 microseconds.

4.5.1 Simple loops

We can perform an exact analysis on a simple loops of the type shown below:

```
i = 0;          /* Statement 1: 1 time */
while(i < n) { /* Statement 2: n+1 times */
   foo();       /* Statement 3: n times */
   i++;         /* Statement 4: n times */
}               /* Loop end ==> implicit GoTo loop start: n times */
```

Letting T_i be the time to perform "statement i" a single time, we can write the equation:

$$T(n) = T_1 + (n+1)T_2 + nT_3 + nT_4 = c_1 n + c_0 = \Theta(n)$$

where $c_1 = T_2 + T_3 + T_4$ and $c_0 = T_1 + T_2$

Let us look at a more general example of a simple loop:

```
for(i = c1; i < n; i = i+c2) /* Assume c1 < n and c2 > 0 */
    foo();   /* Assume foo() takes constant time */
```

How often is the function `foo()` invoked?

We can see that i takes on the values $c_1, c_1 + c_2, c_1 + 2c_2, c_1 + 3c_2, \ldots$ until (but not including) $c_1 + Nc_2 \geq n$. The number of times that `foo()` is called is:

$$\lceil (n - c_1)/c_2 \rceil = \Theta(n)$$

In other words, a loop does not have to do n iterations to be classified as $\Theta(n)$. If the number of iterations is any constant fraction of problem size n, it is $\Theta(n)$.

Loops within loops are also easy to analyze. Consider first:

```
for(i = 0; i < n; i++)
    for(j = 0; j < n; j++)
       foo();
```

Clearly, `foo()` will be invoked n^2 times, so this form of loop is $\Theta(n^2)$.

Now consider a more interesting case:

```
for(i = 0; i < n; i++)
    for(j = i; j < n; j++)
       foo();
```

Here the inner loop executes n times, then $n-1$, then $n-2$ and so on; the last time it executes only once. Hence it does not execute some fixed fraction of n times. While it appears, intuitively, that the complexity of this double loop is $\Theta(n^2)$, how do we prove it?

A simple way to prove this is to note that *on average* the inner loop iterates about $n/2$ times. Since the outer loop is performed n times, `foo()` is invoked about $n^2/2$ times. This is not a rigorous proof, however.

A more precise analysis shows that `foo()` is invoked $\sum_1^n i = n(n+1)/2$ times which is clearly $\Theta(n^2)$.

As a final simple example, consider:

```
for(i = 0; i < n; i++)
    fooN(i);
```

The function `fooN(n)` is known to have $\Theta(n)$ complexity. What is the complexity of the loop?

Once again, the intuitive answer—which also happens to be correct—is $\Theta(n^2)$. How do we prove this?

We see that `fooN()` is invoked n times with the arguments $0, 1, 2 \ldots n-1$. Since the time to perform a function of $\Theta(n)$ complexity can be expressed as $c_1 n + c_0$, the total time spent in the `fooN()` function is:

$$\begin{aligned} T(n) &= c_0 + c_1 \; + \; c_0 + 2c_1 + c_0 \; + \cdots + \; (n-1)c_1 + c_0 \\ &= nc_0 + c_1 \sum_{i=1}^{n} i \\ &= nc_0 + c_1 n(n-1)/2 \\ &= c_2' n^2 + c_1' n + c_0' \end{aligned}$$

Clearly then it is of $\Theta(n^2)$ complexity.

Not every simple `for` loop is $\Theta(n)$ complexity, however. For example, consider:

```
for(f = c1; f < n; f = 2*f)
    foo();   /* Assumed to be of constant time */
```

How often is `foo()` invoked?

In this case, the loop is performed first with $f = c_1$, then with $f = 2c_1$, next with $f = 4c_1$ and so on... until $f = 2^N c_1 \geq n$. Hence the number of times the loop iterates is $\lg(n/c_1)$ and is of $\Theta(\log n)$ complexity.

4.6 Analysis of recursive algorithms

We have already analyzed some recursive algorithms. In Chapter 1 we analyzed merge sort and determined that its running time could be expressed with the recurrence $T(n) = 2T(n/2) + n$. Then, in Chapter 2 we analyzed the Towers of Hanoi algorithm and found that its time was expressed with the recurrence $M(n) = 2M(n-1) + 1$.

Each of these recurrences was solved in closed form (for merge sort we obtained $T(n) = n \lg n$ and for Towers $M(n) = 2^n - 1$). The methods we used were somewhat *ad hoc*; we now look at some other methods.

4.6.1 Solving recurrences

The *unrolling* or *expansion* method of solving a recurrence involves repeating plugging in the recurrence for lower values until a pattern emerges. It is usually a good idea to confirm the pattern using mathematical induction.

For example:

$$\begin{aligned} T(2) &= 2T(n/2) + n \\ &= 2(2T(n/4) + n/2) + n = 4T(n/4) + 2n \\ &= 4(2T(n/8) + n/4) + 2n = 8T(n/8) + 3n \\ &= 8(2T(n/16) + n/8) + 3n = 16T(n/16) + 4n \\ &\vdots \end{aligned}$$

Assuming that $n = 2^N$ and that $T(n/2^N) = T(1) = 0$, we can see that $T(n) = Nn = n \lg n$.

Recurrence trees

A variation on the *unrolling* method uses a visual representation as shown in Figure 4.1 where the recurrence $T(n) = 2T(n/2) + n$ is shown as a diagram and separated into its recursive and non-recursive portions. In this case, to calculate $T(n)$, $T(n/2)$ is calculated twice (the recursive part) and the non-recursive part, n, is added to the final result.

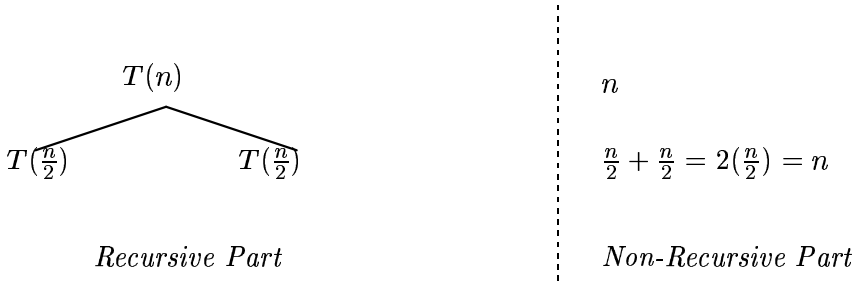


Figure 4.1: Recurrence tree pattern for $T(n) = 2T(n/2) + n$

The power of this visual technique can be appreciated by looking at the recurrence tree for $T(8)$ (where $T(0) = 0$) as shown in Figure 4.2. It is apparent that the contribution of each row in the tree (i.e. the non-recursive additions) is 8 for all rows except the bottom row where we have the base case of $T(0) = 0$. Furthermore, the number of rows is $1 + \lg 8 = 4$. Hence the value of $T(8)$ is 4 (number of rows) $\times$ 8 (contribution per row) $= 32$.

With a little thought, it is apparent that, in general, the contribution of each row in the recurrence tree for $T(n)$ is n and the number of rows in the tree is $1 + \lg n$. Consequently, $T(n) = n(\lg n + 1) = n \lg 2n$ in general.

Just in case the previous paragraph needs more than "a little thought", Figure 4.3 shows the general behavior. Once the general pattern is clear—in this case, the fundamental thing to note is that the non-recursive contribution of each row is n (except for the $T(0)$ row)—the next tricky thing to determine is an expression for the number of rows. In this case, it is $\lg n + 1$ (*not* $\lg n$). It is easy to get this wrong (and I have done so at the blackboard in class...). If you draw draw a specific recurrence tree (eg. $T(8)$) as well as the general

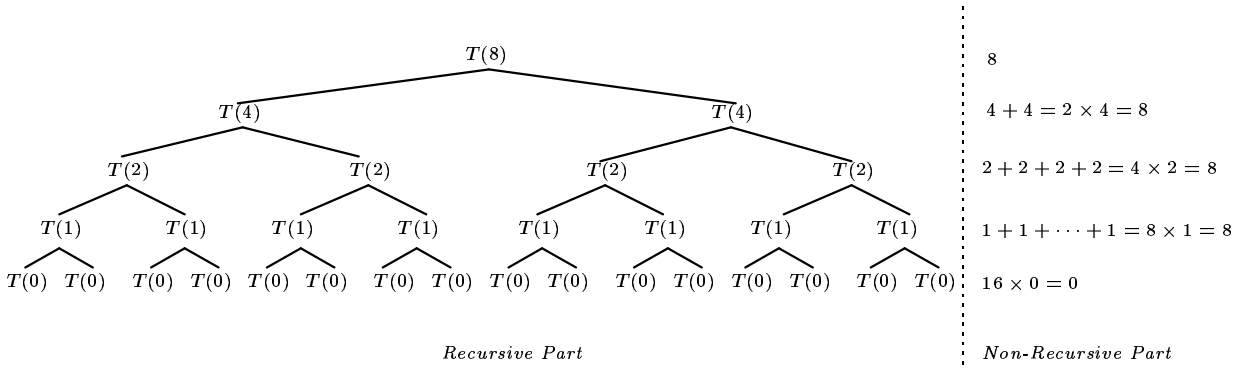


Figure 4.2: Recurrence tree for $T(n) = 2T(n/2) + n$ where $n = 8$

tree for $T(n)$, you can confirm your general conclusions with reference to both trees.

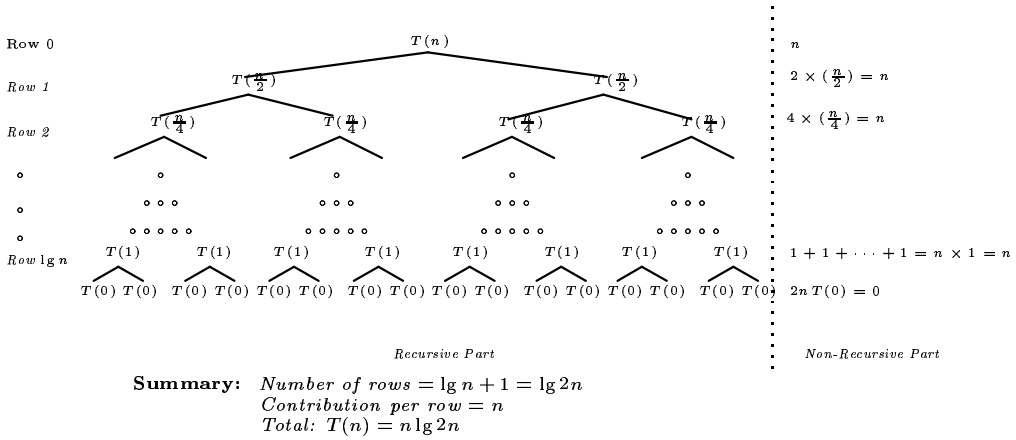


Figure 4.3: Recurrence tree for $T(n) = 2T(n/2) + n$—general case

Now that we have analyzed the recurrence $T(n) = 2T(n/2) + n$ in many different ways (starting in Chapter 2), let's see how we can apply this recurrence visualization technique to other problems.

Version 1.1 (2003-03-11) (Chapter version: 2003-03-11)

Solving T(n) = 3T(n/2) + n

As our first example, consider the recurrence:

$$T(n) = \begin{cases} 0 & \text{if } n = 0 \\ 3T(n/2) = n & \text{otherwise} \end{cases}$$

Before trying to obtain a closed-form solution to this recurrence, let's calculate by hand the first few values by directly using the recurrence. We do this only for values of n that are exact powers of 2 (i.e. $n = 2^i$) so that we can always divide by 2 and get an exact integer result. The manually calculated values are shown in Table 4.1.

$\lg n$	n	$T(n) = 3T(n/2) + n$
–	0	0
0	1	3T(0)+1=1
1	2	3T(1)+2=5
2	4	3T(2)+4=19
3	8	3T(4)+8=65
4	16	3T(8)+16=211
5	32	3T(16)+32=665

Table 4.1: $3T(n/2) + n$ for selected values assuming $T(0) = 0$

The method suggested in Chapter 2 to solve recurrences—i.e "Guess the closed-form solution and prove it by mathematical induction"—does not seem appropriate here. How do we "guess" the answer? Few people could look at the sequence 0, 1, 5, 19, 65, 211, 655 ...and "see" the "obvious" pattern in the blink of an eye!

However, we can often discover the pattern and solve the problem using the recurrence tree visualization technique.

First, let's draw the recurrence tree for a specific instance of the problem: $T(8)$. Figure 4.4 shows the result.

When we look at the tree, we notice:

- The contribution of each row is not a simple constant. Row 0 contributes 8, row 1 contributes 12, row 2 contributes 18...
- Nonetheless, we *can* see that the number of contributing rows is 4.

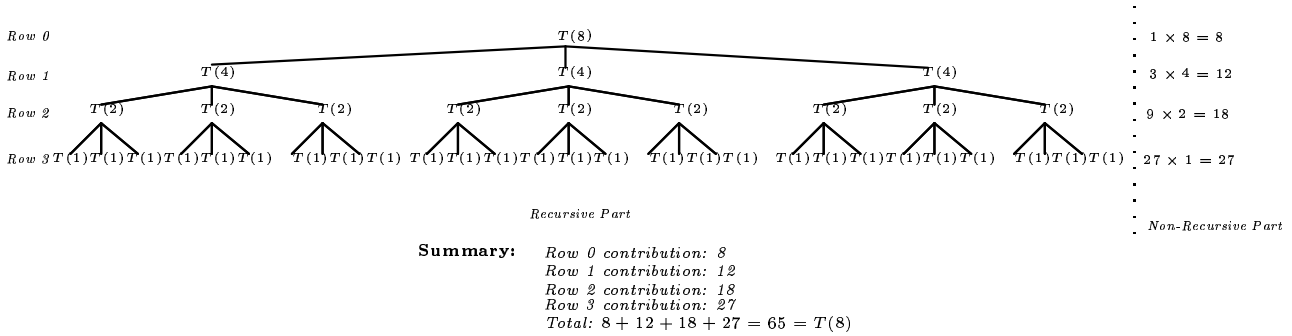


Figure 4.4: Recurrence tree for $T(n) = 3T(n/2) + n$ where $n = 8$

Now we draw the recurrence tree (partially) for the general problem: $T(8)$. Figure 4.5 shows the result.

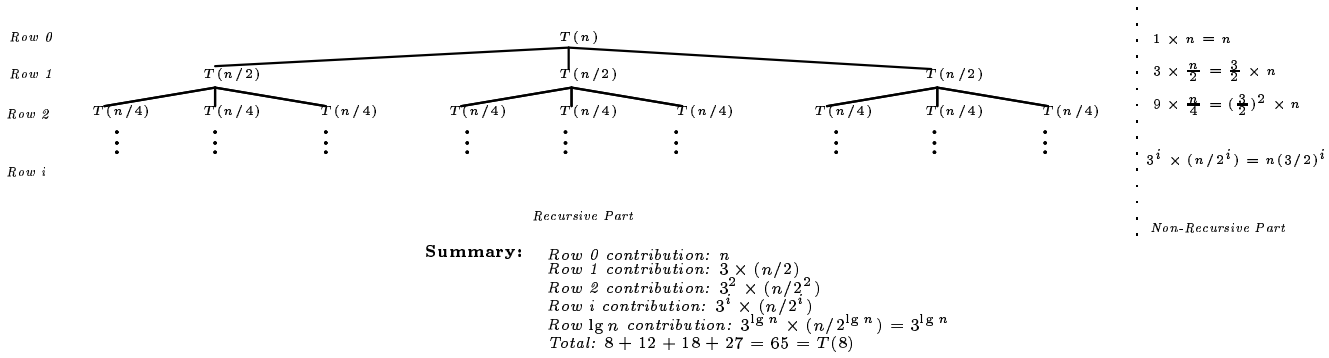


Figure 4.5: Recurrence tree for $T(n) = 3T(n/2) + n$

We can now see some general characteristics of the recurrence tree:

- As before, the contribution of each row is not a constant. The number of nodes in the recurrence tree is multiplied by 3 for each deeper row.
- In general, the number of nodes in Row i is 3^i.

Version 1.1 (2003-03-11) (Chapter version: 2003-03-11)

- The contribution of each node in any particular row is the same. However, this contribution is not the same for each row; indeed, it is halved each time we go to a deeper row.
- In general, then, we have:

$$\begin{aligned} \text{Row}_i \text{ contribution per node} &= n/2^i \\ \text{Row}_i \text{ number of nodes} &= 3^i \\ \text{Row}_i \text{ total contribution} &= n/2^i \times 3^i = (\frac{3}{2})^i n \end{aligned}$$

- Finally we note that the total number of rows is $\lg n + 1$.

Consequently, we can obtain the value for $T(n)$ by adding up the total contributions of each row:

$$T(n) = \sum_{i=0}^{\lg n} (\frac{3}{2})^i n$$

This is a simple geometric series. Recall (from high school algebra) that:

$$1 + x + x^2 + \cdots + x^k = \sum_{i=0}^{k} x^i = \frac{x^{k+1} - 1}{x - 1}$$

In this case, then, we obtain:

$$T(n) = n\frac{(3/2)^{\lg n+1} - 1}{(3/2) - 1} = 2n((3/2)^{\lg n+1} - 1) = 3^{\lg n+1} - 2^{\lg n+1}$$

Is this the right answer? Yes...if no mistakes have been made.

We can do a quick check by using the closed form equation to calculate $T(32)$, where $\lg 32 = 5$. We obtain:

$$2 \times 32(3^6/2^6 - 1) = 2^6(\frac{3^6 - 2^6}{2^6}) = 3^6 - 2^6 = 729 - 64 = 665$$

Fortunately, this gives the same answer as the manually calculated one based directly on the recurrence as shown in Table 4.1.

This is good circumstantial evidence that the equation is correct. If you are unconvinced, you can always prove it using mathematical induction.

A Simple Example

We have now illustrated the recurrence tree visualization technique for two cases. In both cases, the number of nodes in any row increased as the recurrence tree became deeper. We now turn our attention to simpler recurrences where the number of nodes at each level is constant.

First, recall the simple example of recursion—`add(pink,blue)`—discussed in Chapter 2. The pink-blue algorithm expressed the idea of *addition* using only simple counting operations. Specifically, the only operations used were "increment/decrement by one". The algorithm could have been expressed as the following recurrence:

$$add(pink, blue) = \begin{cases} blue & \text{if } pink = 0 \\ add(pink - 1, blue + 1) & \text{otherwise} \end{cases}$$

We already know, of course, that the closed-form solution to `add(pink, blue)` is simply `pink+blue`.

We want to consider, however, a variation on the recurrence:

$$foo(pink, blue) = \begin{cases} blue & \text{if } pink = 0 \\ foo(pink - 1, blue + 1) + 1 & \text{otherwise} \end{cases}$$

What is the closed-form expression for `foo(pink, blue)`?

Perhaps you can just "see" the answer as "obvious". But let's pretend that you can't and use the recursion tree visualization technique to solve this problem.

First, however, we build a table manually for some simple instances of the function $foo()$. Table 4.2 is an example. (Note that the table was filled in so that no row required only increment and decrement operations on numbers or previously calculated values.)

Even though you might now be tempted to guess the closed-form expression for $foo(pink, blue)$, let us soldier on and draw a recurrence tree for a specific instance. We choose to do this for $foo(6, 9)$ as shown in Figure 4.6.

We note that the contribution of each row is 1 except for the last row where the contribution is 15. There are 7 rows in all. Thus the value of $foo(6, 9)$ is $6 + 15 = 21$.

At this point, the overall pattern can be discerned even without drawing a generalized recurrence tree. There will be $pink + 1$ rows numbered $0 \cdots pink$. All of the rows numbered $0 \cdots pink - 1$ contribute 1 while row $pink$ contributes $pink + blue$.

$pink$	$blue$	Derivation	$foo(pink, blue)$
0	x	$foo(0, x) = x$	x
1	0	$foo(1, 0) = foo(0, 1) + 1 = 2$	2
1	1	$foo(1, 1) = foo(0, 2) + 1 = 2 + 1 = 3$	3
1	2	$foo(1, 2) = foo(0, 3) + 1 = 3 + 1 = 4$	4
2	1	$foo(2, 1) = foo(1, 2) + 1 = 4 + 1 = 5$	5

Table 4.2: $foo(pink, blue)$ for selected values

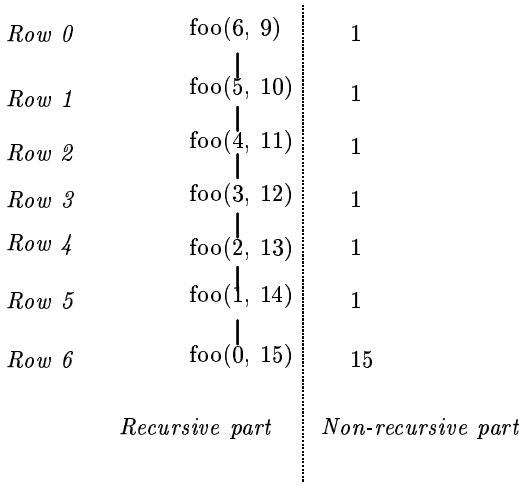


Figure 4.6: Tree for $foo(pink, blue) = foo(pink - 1, blue + 1) + 1$ where $pink = 6, blue = 9$

Consequently, we have:

$$foo(pink, blue) = \underbrace{pink \times 1}_{pink\ rows\ each\ contributes\ 1} + \underbrace{(pink + blue)}_{last\ row\ contributes\ pink+blue} = 2 \times pink + blue$$

4.6.2 Generating functions (z-transforms)

One very general way for solving recurrences is to use generating functions.

The generating function of a sequence f_i is defined as:

$$G(z) = \sum_i f_i z^i$$

(Engineers use a very similar concept—the *z-transform*—which assumes that the sequence is "signal values" sampled at equal time intervals. For these sequences, the z-transform is defined as $Z(f) = \sum_i f_i z^{-i}$. Although very similar, there are syntactical and semantic differences between generating functions and z-transforms, however.)

For the Fibonacci series, the generating function is:

$$F(z) = z + z^2 + 2z^3 + 3z^4 + 5z^5 + 8z^6 + 13z^7 + \ldots$$

Evaluating $z + zF(z) + z^2F(z)$, we obtain:

$$\begin{aligned}
z &= z \\
zF(z) &= z^2 + z^3 + 2z^4 + 3z^5 + 5z^6 + 8z^7 + 13z^8 + \ldots \\
z^2F(z) &= z^3 + z^4 + 2z^5 + 3z^6 + 5z^7 + 8z^8 + 13z^9 + \ldots \\
\Rightarrow z + zF(z) + z^2F(z) &= z + z^2 + 2z^3 + 3z^4 + 5z^5 + 8z^6 + 13z^7 + \ldots \\
&= F(z)
\end{aligned}$$

Consequently:

$$\begin{aligned}
F(z) &= \frac{z}{1 - z - z^2} \\
&= \frac{z}{(1 - \phi z)(1 - \hat{\phi} z)} \\
&= \frac{1}{\sqrt{5}} \left(\frac{1}{1 - \phi z} - \frac{1}{1 - \hat{\phi} z} \right)
\end{aligned}$$

where:

$$\phi = (1 + \sqrt{5})/2 = 1.61803\ldots \text{ and } \hat{\phi} = (1 - \sqrt{5})/2 = -0.61803\ldots$$

Hence:

$$F(z) = \sum_{i=0}^{\infty} \frac{1}{\sqrt{5}} (\phi^i - \hat{\phi}^i)$$

But, by definition, the ith term in the infinite series is the ith Fibonacci number. Hence:

$$Fib(i) = \frac{1}{\sqrt{5}} (\phi^i - \hat{\phi}^i)$$

Version 1.1 (2003-03-11) (Chapter version: 2003-03-11)

Since $\hat{\phi} < 1$, we also have:

$$Fib(i) = \lfloor \frac{\phi^i}{\sqrt{5} + 0.5} \rfloor$$

Consequently, the Fibonacci series grows exponentially.

4.7 Problems

4.1 The following functions represent the running times (in some convenient unit of time) of different algorithms as a function of problem size n.

1. $$n^2 + n + 5$$

2. $$200n + 6$$

3. $$\lg n!$$

1. Determine the Θ complexity of each function and rank them from fastest to slowest for asymptotically large values of n. (Indicate a tie if more than one function has the same Θ complexity.)

2. Rank the functions from fastest to slowest if the value of n is 100.

3. What is the smallest value of n such that the ranking of the actual running times corresponds to the ranking using Θ complexity?

4.2 An algorithm requires $T(n) = 2.6n^3 + \lg n^{10} + 123.456$. Which of the following statements are true:

1. $T(n)$ is $O(n^6)$.
2. $T(n)$ is $\Omega(n^6)$.
3. $T(n)$ is $\Theta(n^6)$.
4. $T(n)$ is $O(n^3)$.
5. $T(n)$ is $\Omega(n^3)$.
6. $T(n)$ is $\Theta(n^3)$.

7. $T(n)$ is $O(n^2)$. 8. $T(n)$ is $\Omega(n^2)$. 9. $T(n)$ is $\Theta(n^2)$.

4.3 Show that for any positive integer k:

$$\sum_{1 \le i \le n} i^k = \Theta(n^{k+1})$$

4.4 Show that

$$\log\left(K_1 n! + P_k(n)\right) = \Theta(n \log n)$$

where K_1 is an arbitrary constant and $P_k(n)$ is an arbitrary polynomial of degree k and k is some positive integer.

4.5 For the following type of loop:

```
float f;
for(f = c1; f < n; f *= c2)
   foo()  /* constant time */
```

Determine the necessary conditions so that the loop will execute a finite number of times.

Develop an exact expression for the number of times the loop iterates. What is its $\Theta()$ complexity?

4.6 In Chapter 1 (page 12) we derived:

$$T(n) = T_{split} + T(n/2) + T(n/2) + K_1 n + K_0 \tag{4.1}$$

This was simplified by setting $T_{split} = 0$, $K_0 = 0$ and $K_1 = 1$ to obtain:

$$T(n) = 2T(n/2) + n \tag{4.2}$$

With these simplifications, we showed that $T(n) = n \lg n$ (with $T(1) = 0$). Prove that the original equation rewitten as $T(n) = 2T(n/2) + an + b$ where $T(1) = c$ is $\Theta(n \log n)$.

i.e.:

$$T(n) = \begin{cases} c & \text{if } n = 1 \\ 2T(n/2) + an + b & \text{otherwise} \end{cases}$$

4.7 Prove that $n \log n$ grows more slowly than $n^{1+\epsilon}$ where $\epsilon > 0$.

Chapter 5

Abstract Data Types

> *Everything should be made as simple as possible, but not simpler.*
>
> —Albert Einstein

> *Let's look under the hood.*
>
> —Anonymous

This chapter examines the concept of *Abstract Data Types* (ADT) and how these concepts can be implemented in a procedural language like C.

5.1 What is an ADT?

A simple definition of an ADT is:

> A data type and an associated set of operations (defined as the type's *interface*) that can be performed with the type.

We say that the interface *encapsulates* the ADT. The documentation of the interface is all a programmer who wants to use an ADT needs to know. The actual inner workings of the operations or the layout of the items in the data type should be of no concern to the application programmer using the ADT. Indeed, it is preferable to *hide* the information, making it impossible for the user to exploit implementation details; users should be forced to manipulate the ADT *only* with the publicly documented interface.

In this chapter we will examine ADTs from two perspectives: the application programmer's (user) point of view as well as that of the implementor

(the ADT programmer). We will look at several examples; in each case we first examine the ADT from the user perspective and then look at how the implementor created it.

The concepts explored here are applicable to any programming environment from assembly language programming on "bare metal" to object-oriented programming in languages like Java, C++ or Smalltalk. The primary implementation language used here, however, is C. This makes some of the concepts harder to implement than they would be in Java; but it has the advantage of demonstrating how some of the "smoke and mirror" tricks of object-oriented languages are achieved.

The advantages of using ADTs include greater software reliability, more flexibility and the possibility of re-using existing software more easily. For example, in Chapter 1 (page 15), we stated "It would be nice to be able to sort any kind of data with a single sort routine" but we did not show how to do it then. In this chapter, we will look at ways to solve this kind of problem.

5.2 A Simple "Bag" ADT specification

When you shop at a grocery store, you place each item, one at a time, into your bag. Upon returning home, you then remove each item, one at a time, from the bag. If you understand this simple analogy, you understand what we mean by a "bag" in this chapter. It should be noted, however, that the contents of this kind of bag are unordered; i.e. the order that the items are placed in the bag is not necessarily related to the order in which they are removed. We will also idealize our conceptual or abstract bag in one more way—we assume that it has unlimited capacity.

A *bag* is defined mathematically as an unordered collection of elements. The only operations we can perform on a bag is to *add* an item or *remove* an item[1].

The basic operations that can be performed on a Bag are:

add Add something to a bag.

remove Take an item out of a (non-empty) bag.

[1] In mathematics, other set operations such as union and intersection or finding an item in a bag are often considered as well. We do not consider these operations, however.

In addition, we need an operation to create an empty bag in the first place. As a convenience, we also add an operation that tells us how many items are in a bag:

new Create a new empty bag.

getSize Determine how many items are in a bag.

There may also be another operation required—*destroy*—for getting rid of a bag once we have finished with it. In everyday life, we are usually responsible for getting rid of things we no longer need by "throwing them in the garbage." Sometimes, however, we can be much more irresponsible: when we vacate a hotel room, for example, we can leave things scattered all over. The cleaning staff will collect our garbage and (hopefully) return important items inadvertently left behind.

Some programming languages also "provide a cleaning staff" that takes care of our mess. Whether or not we need a destroy operation depends on the programming environment the abstract notion of Bag is implemented in. In C or C++, such an operation is usually required; in Java or lisp, it is usually not required (bags are picked up and thrown away automatically when they are no longer needed by the "garbage collector".)

5.3 Basic implementations of an "integer bag"

We now look at how to implement the basic idea of a Bag in C. For simplicity, we first consider a specialized version of a Bag that contains only integers.

A Bag is clearly a collection of things, and collections are usually implemented as either arrays or lists. We will start by using a linked list implementation since this is the easiest way to create a bag that that has virtually unlimited capacity and does not use more memory than is required.[2]

Later, we will use a type of array, but not the simple arrays that are directly supported in the C language. (The problem with these arrays is that their size is fixed at compile time which leads to bag sizes that either ridiculously large or ones that are too small to be considered as having "unlimited size".)

[2]The memory requirements are $\Theta(n)$, more precisely $2n + 2$ "units" of memory where each "unit" is the space required for an `int`, usually 4 bytes.

5.3.1 Linked list implementation of IntBag

The first concrete implementation of the abstract concept of Bag restricts the items in the Bag to `int`s only and uses a linked list to maintain them. To remind us of these restrictions and specializations, we call this kind of Bag a `IntLLBag`.

One of the hallmarks of a modularized design is that a programmer should be able to use an ADT given only the API (Application Programming Interface) documentation; there should be no need to have access to the source code or use any special knowledge about the implementation.

To use the IntLLBag ADT, the application programmer is told to include the file `IntLLBag.h` (containing function prototypes and `typedef`s) and link the object file `IntLLBag.o` with their application. The API for the IntLLBag is:

IntLLBag newIntLLBag(void) Returns a newly created *IntLLBag* or `NULL` if one cannot be created.

void addIntLLBag(IntLLBag b, int i) Adds the integer i to the specified bag b.

int removeIntLLBag(IntLLBag b) Removes an integer (and returns it) from the specified bag b. The program exits if the bag is empty. Note that the order of removal is *not* specified.

unsigned int getSizeIntLLBag(IntLLBag b) Returns the number of items in bag b.

void destroyIntLLBag(IntLLBag b) Destroys a previously created IntLLBag, releasing all its resources.

With this knowledge, a simple program using an `IntLLBag` can be written:

```
#include "IntLLBag.h"

int main(int argc, char * argv[])
{
  IntLLBag b;
```

```
  b = newIntLLBag();

  addIntLLBag(b, 3);
  addIntLLBag(b, 1);
  addIntLLBag(b, 4);
  addIntLLBag(b, 1);

  while(getSizeIntLLBag(b))
      printf("Removed: %d\n", removeIntLLBag(b));
  exit(0);
}
```

With the implementation we are about to discuss, the numbers will be removed in the opposite order they were added (i.e. they will be removed in the order 1, 4, 1, 3.) It is essential to understand, however, that the application programmer *must not rely* on this behavior. Indeed, the API specifically warns the programmer about this.

The reason this is so important is that it allows the implementation of the ADT to be radically changed (so long as it still conforms the API) and the second implementation to be used interchangeably with the first. This is an example of *encapsulation* .

(If the application programmer really needs an ADT where the order of removal is well-defined—e.g. first in, first out—they should choose an ADT that guarantees this kind of behavior.)

IntLLBag implementation

We now look "under the hood" at the source code to implement the IntLLBag abstract data type. We want to write the code so that the implementation is well encapsulated, corresponds to the public API and hides all details about the implementation from the user of the ADT. To achieve these aims in a language like C requires some coding conventions that are not often used in more elementary programming.

The application programmer must, of course, have access to the public header file `IntLLBag.h`. Amongst other things, this header must define the `IntLLBag` data type, and it must do so in such a way that implementation details are hidden (i.e. information hiding.) This trick is accomplished by using an "opaque" or generic data type as follows (taken from `IntLLBag.h`):

```
typedef void * IntLLBag;
```

The application programmer might guess that the "real" data type is a pointer to a data structure, but they should avoid such speculation. The application programmer does not need to know the inner workings.

The remainder of the header file contains function prototypes and documentation for the various operations than can be performed with an `IntLLBag`. For example:

```
/** Add an integer to an IntLLBag.
 *
 *  @param b The Bag that will be added to.
 *  @param i The integer to add.
 */

void addIntLLBag(IntLLBag b, int i);
```

So far we have not really looked under the hood, since the header file, although written by the ADT implementor, is not hidden from the application programmer. Let's now look at the actual implementation source code—`IntLLBag.c` which *is hidden* from the user.

Towards the beginning of the source code is the following data structure definition:

```
typedef struct _LList _LList, *_LListPtr;
struct _LList {
   int data;
   _LListPtr next;
};
```

This defines the basic data structure for nodes on a linked list. Each item on the list contains integer data (after all, it is a bag of integers) and a pointer to the next item on the list. Note that the name of the data type (`_LList`) begins with '_' (the underscore character); this programming convention indicates that the data type is private to the implementation and the user (the application programmer) should have no knowledge of these private data structures. This means, for example, that such data types would not appear as arguments or return values from any of the publicly documented functions. Using the '_' convention serves as a reminder to the

ADT implementor that application programmers should never be exposed to these private data structures and that the implementor can fiddle with them in any way they like (so long as the interface documentation is not compromised.)

The next private data type specifies how an `IntLLBag` is actually laid out:

```
typedef struct _IntLLBag _IntLLBag, *_IntLLBagPtr;
struct _IntLLBag {
  _LListPtr head;
  int size;
};
```

The basic structure consists of a pointer to the first item on the linked list (when there is nothing in the list, this is a NULL pointer). The second field keeps track of the size of the list. (The fact that this field exists is an implementation detail and, indeed, is not necessary. For example, when a count of the number of items on the list is required, we could simply start at the `head` and follow the `next` pointers until we hit a NULL pointer. By keeping track of how many pointers we follow, the size can be determined. This is quite inefficient—it is a $\Theta(n)$ method. By keeping track of the size in a separate field, determining the size becomes a $\Theta(1)$ operation.)

Before looking at the actual source code, Figure 5.1 shows how the private data structures are linked to implement a bag containing two integers. The application programmer, of course, is unaware of what is going on to the right of the vertical dashed line. (They know that `b` is a generic pointer.)

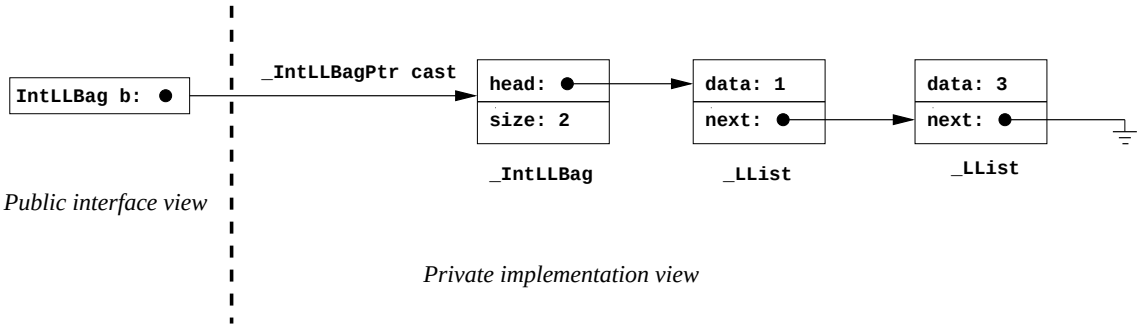


Figure 5.1: Data structures after adding "3" and "1" to empty Bag

Once we have defined these private data structures, the coding for the functions is relatively straight forward. First, we look at the code to create a new `IntLLBag`: the memory for the private structure is allocated and initialized as shown below.

```
IntLLBag newIntLLBag(void)
{
  _IntLLBagPtr b;
  b = malloc(sizeof(_IntLLBag));
  if (b == NULL)
    return NULL;
  b->head = ( _LListPtr )  NULL;
  b->size = 0;
  return (IntLLBag) b;
}
```

To add an integer to a bag, we allocate memory for a node on the list, insert it at the beginning of the list (so it is now the head of the list) and set its data field the value of the integer being added, and increment the `size` field of the private Bag data structure to reflect the fact that there is now one more item in the bag.

The code is shown below:

```
void addIntLLBag(IntLLBag b, int i)
{
  _LListPtr item;
  _IntLLBagPtr _b = (_IntLLBagPtr) b;

  item = malloc(sizeof(_LList));
  item->data = i;
  item->next = _b->head;
  _b->size++;
  _b->head = item;
  return;
}
```

Note the line:

```
_IntLLBagPtr _b = (_IntLLBagPtr) b;
```

This cast is essential because the `IntLLBag b` parameter that is passed is an opaque data structure that the compiler does not "know" has fields called "size" and "head". We do not really need the private local variable `_b` and could have achieved the same result with explicit casts of parameter `b` as follows:

```
((IntLLBagPtr)b)->size++;
```

This is somewhat uglier code, however: it is more tedious and harder to read and write. Furthermore, a good compiler will probably "optimize away" the local variable `_b`.

We next look at the code for removing an item from a bag. First, we check that the bag has at least one item in it; if it is empty, we unceremoniously exit. (This extreme measure would almost certainly be inappropriate in production quality code; these examples, however, are meant only to be illustrative of programming techniques for ADTs.) Assuming there is something to remove, we unlink it from the list (and we take the first one in the list), reset the "head" field and decrement the size. We then extract the integer value form the unlinked node, which will be returned. Before returning, however, and *very importantly*, the memory used by the node's data structure is released. If we don't do this, a memory leak will be created. (This is clearly the responsibility of the ADT implementation; after all, the application programmer is not even aware of these private data structures and certainly has no way of releasing the memory resources consumed by them.)

The code is shown below:

```
int removeIntLLBag(IntLLBag b)
{
  int r;
  _LListPtr item;
  _IntLLBagPtr _b = (_IntLLBagPtr) b;

  if (_b->size <= 0) {
    eprintf("Fatal error, removing from empty bag\n");
  }
  _b->size--;
```

```
  item = _b->head;
  r = item->data;
  _b->head = item->next;
  free(item);
  return r;
}
```

To destroy an `IntLLBag`, we remove all the items until the bag is empty and then release the memory for the `_IntLLBag` data structure. The code is:

```
void destroyIntLLBag(IntLLBag b)
{
  _IntLLBagPtr _b = (_IntLLBagPtr) b;

  while(_b->size > 0) {
    removeIntLLBag(b);
  }
  free(_b);
  return;
}
```

Note that the implementation of `destroyIntLLBag` invokes the public interface function `removeIntLLBag` so this part of the destruction procedure could have been done by the application programmer. However, freeing the memory for the private data structure should only be done by the ADT implementation code.

Finally, the implementation of the `getSizeIntLLBag()` function is trivial:

```
unsigned int getSizeIntLLBag(IntLLBag b)
{
   return ((_IntLLBagPtr)b)->size;
}
```

5.3.2 Resizeable array implementation of an Integer Bag

We now look at a different implementation of a Bag of integers using a resizeable array technique instead of a linked list. As noted previously, we

cannot use an ordinary C array because its size is fixed at compile time and we do not know in advance how big to make the array so that it appears to have "unlimited size" to the application programmer.

(One of the problems at the end of the chapter does suggest a way that fixed size arrays could be used if the application programmer could specify at bag creation time the maximum size that a specific bag will ever have to be.)

In this section we show how resizeable arrays can be achieved in C and use this technique to implement the Bag ADT. Resizeable arrays are sometimes called *Vectors*, so we will call this implementation a `IntVBag`.

From the application programmer's point of view, the interface to the `IntVBag` is identical to that for the `IntLLBag` (except, of course, that every occurrence of "IntLLBag" becomes "IntVBag".) Indeed, the simple test program shown on page 104 could be rewritten to use an `IntVBag` simply by replacing each occurrence of "LL" with "V" in the source code.

The public header file—`IntVBag.h`—is very similar to `IntLLBag.h`. Function prototypes are given and an opaque data type is defined for the application programmer:

```
typedef void * IntVBag;
```

As in the case of the linked list implementation, we start the `IntVBag.c` source code file with a private data type that will contain the bag:

```
typedef struct _IntVBag _IntVBag,* _IntVBagPtr;
struct _IntVBag {
  int * data;
  int size;
  int maxSize;
};
```

As in the case for the linked list implementation, the `size` field keeps track of the number of integers in the bag. The field `maxSize` indicates how much memory is currently allocated for the integers in the bag. The `data` field is a pointer to allocated memory for the data in the bag. It is declared as a pointer to an integer, but the implementation will often use it as the name of an array. Suppose, for example, that `data` pointed to a chunk of memory that was big enough to hold 8 integers. In that case, the value of

`maxSize` would be 8 and and reference to `data[i]` where `i` was between 0 and 7 (inclusively) would be legal.

Once again, Before looking at the actual source code, Figure 5.2 shows how the private data structures are linked to implement a bag containing three integers.

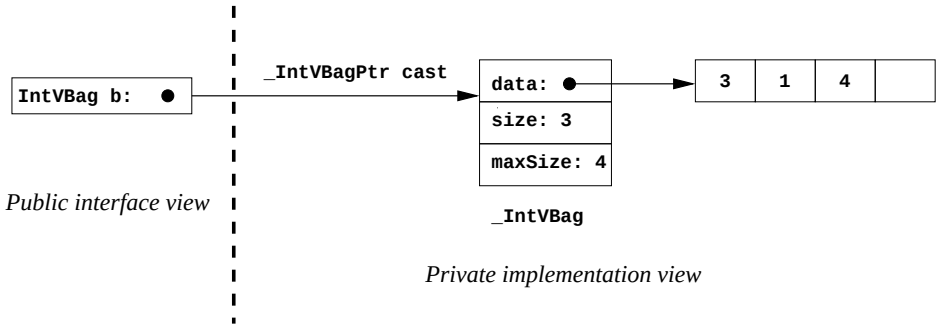


Figure 5.2: Data structures after adding 3, 1 and 4 to empty IntVBag

Creating a new empty bag is straight forward:

```
IntVBag newIntVBag(void)
{
  _IntVBagPtr b;

  b = malloc(sizeof(_IntVBag));
  if (b == NULL)
    return NULL;
  b->size = 0;
  b->maxSize = 0;
  b->data = NULL;
  return (IntVBag) b;
}
```

The first interesting code is `addIntVBag`.

First consider a simple case: suppose that `maxSize` is 8 (and hence that there is memory allocated for 8 integers) and that there are currently only 4 integers in the bag. We can access any of these with `b->data[i]` where `i` is 0, 1, 2, or 3. Since there are 4 unused slots in the resizeable array, we can simply add the new integer at `b->data[4]` and increment the `size` field.

The more interesting case occurs when there is not enough room in the memory chunk referred to by `data` to hold another integer. This case occurs when `maxSize==size`, but in two slightly different ways: either `data` points "nowhere" (i.e. `maxSize` = 0) or some memory has already been allocated but is full (i.e. `maxSize>0`). In the former case, we are adding the first integer to a newly created bag; we allocate enough memory to hold exactly one integer, copy it there and increment `maxSize` and `size`. Otherwise, we expand the size of the memory chunk containing the integers by a factor of two.

This is done using the ANSI standard library function `realloc()`. The function prototype (from `<stdlib.h>`) is:

```
void *realloc(void *ptr, size_t size);
```

The on-line manual for standard library C functions describes `realloc()` as:

> realloc() changes the size of the block pointed to by ptr to size bytes and returns a pointer to the (possibly moved) block. The contents will be unchanged up to the lesser of the new and old sizes. If ptr is NULL, realloc() behaves like malloc() for the specified size. If size is zero and ptr is not a null pointer, the object pointed to is freed.

This is a very handy memory allocation function and is often very efficient. To understand why, you need a bit of knowledge about how memory allocation works in cooperation with an operating system(OS). When memory is obtained with `malloc()`, there is a reasonable chance that the chunk of memory allocated is immediately followed by another block of memory that is not being used and is available for extending the size of the chunk initially allocated. When this happy circumstance arises upon a call to `realloc()`, the cost (in time) of increasing the size of the block is virtually nothing. The more expensive operation of actually allocating a new (and bigger) chunk of memory and copying the contents of the previous block to the new one occur only "some" of the times.

The code for `addIntVBag` is:

```
void addIntVBag(IntVBag b, int i)
{
```

```
  _IntVBagPtr _b = (_IntVBagPtr) b;

  if (_b->size >= _b->maxSize) { /* Not enough room? */
    if(_b->maxSize == 0) {
        _b->maxSize =  1;  /* Make space for 1 if now empty */
    } else {
        _b->maxSize *= 2; /* Otherwise, double "array" size */
    }
  _b->data = realloc(_b->data, _b->maxSize*sizeof(int));
  }
  _b->data[_b->size] = i;
  _b->size++;
  return;
}
```

Deleting an integer from a bag is very simple (simpler than the linked list implementation). We simply decrement the `size` field of the private data structure and return the integer that was at the end of the array. The code is:

```
int removeIntVBag(IntVBag b)
{
  _IntVBagPtr _b = (_IntVBagPtr) b;

  if (_b->size <= 0) {
    eprintf("Fatal error, removing from empty bag\n");
  }
  return _b->data[--_b->size];
}
```

Destroying an `IntVBag` is also simple: release the memory for the data and for the data structure as shown below.

```
void destroyIntVBag(IntVBag b)
{
  _IntVBagPtr _b = (_IntVBagPtr) b;
  free(_b->data);
  _b->size = 0;
```

```
  _b->data = NULL;
  free(_b);
  return;
}
```

As before, determining the size of a bag is trivial:

```
unsigned int getSizeIntVBag(IntVBag b)
{
  _IntVBagPtr _b = (_IntVBagPtr) b;
  return _b->size;
}
```

5.4 IntBag

The previous two implementations of "integer bags" are not really abstract enough. In particular, the application programmer is aware of how they are implemented by their very names. More importantly, if the programmer initially wrote an application using `IntLLBag`s and then decides that `IntVBag`s would be superior in some cases, she has to manually go through her source code and change function names like `addIntLLBag` to `addIntVBag` which is boring, tedious and error-prone.

The interface to `IntBag` is identical to the ones for `IntVBag` and `IntLLBag` except that no hint is given about the method of implementation (i.e. perhaps it is with a resizeable array, perhaps a linked list, or perhaps something completely different.)

The interface is:

IntBag newIntBag(void) Returns a newly created *IntBag* or `NULL` if one cannot be created.

void addIntBag(IntBag b, int i) Adds the integer i to the specified bag b.

int removeIntBag(IntBag b) Removes an integer (and returns it) from the specified bag b. The program exits if the bag is empty. Note that the order of removal is *not* specified.

unsigned int getSizeIntBag(IntBag b) Returns the number of items in bag *b*.

void destroyIntBag(IntBag b) Destroys a previously created IntBag, releasing all its resources.

In addition to the basic interface, the application programmer can also explicitly choose to use a vector or linked list implementation of the bag by using the following interface:

IntBag newIntLLBag(void) Returns a newly created *IntBag* or `NULL` if one cannot be created. The *IntBag* will be implemented using a linked list.

IntBag newIntVBag(void) Returns a newly created *IntBag* or `NULL` if one cannot be created. The *IntBag* will be implemented using a vector.

Irrespective of the specific type of IntBag created, the programming interface is identical. Thus the application programmer need only change the "new" function when a different implementation of a bag is desired.

The interface specification can be used by the application programmer as follows:

```
#include "IntBag.h"
int main(int argc, char * argv[])
{
  IntBag b1, b2;

  b1 = newIntBag(); b2 = newIntLLBag();

  addIntBag(b1, 3);  addIntBag(b2, 3);
  addIntBag(b1, 1);  addIntBag(b2, 1);
  addIntBag(b1, 4);  addIntBag(b2, 4);
  addIntBag(b1, 1);  addIntBag(b2, 1);

  while(getSizeIntBag(b1))
    printf("Removed: %d (from b1) and %d (from b2)\n",
           removeIntBag(b1), removeIntBag(b2));
  exit(0);
}
```

5.4.1 Under the hood of *IntBag*

The public header file `IntBag.h` is similar to the ones we have already seen.

But, there is *another private* header file as well—`IntBagP.h`—that is used only by the implementor and is hidden from the user.

An examination of this file gives some clues about how an `IntBag` is implemented as well as some coding conventions that are used by the implementor.

First, note the following near the start of the file:

```
#define private static
#define public
```

The word "static" has two meanings in C. When applied to a local variable, it means the memory for the variable is allocated permanently (just like a global variable) and that its value persists from one invocation of a function containing it to another. However, when "static" is used to qualify something of global scope, it means that the qualified name will occupy a private name-space visible only within the same source code file—i.e. the name will not be made available to the linker. In other words, this latter use of the "static" qualifier makes function or global variable names "private". Because the use of static in these files corresponds to the "private" semantics and because the word private is used in many object-oriented languages, I have used the `#define private static` preprocessor directive[3]

For symmetry, I have also defined `public` as "nothing" since global scope is the default for top-level declarations.

Next look at the following somewhat complex `typedef`:

```
typedef void * (*_method)(IntBag b, ...);
```

The new type that is defined here is `_method` and it it a pointer type, specifically a pointer to a function. The function it points to has at least one argument (and the first argument must be an `IntBag`) and it returns a generic pointer (which can be case to any kind of data pointer or to an `int`).

Before looking any further at `IntBag.h` let us briefly look at how pointers to functions can be used. (Readers familiar with pointers to functions in C can skip the following.)

[3] As a rule, I thoroughly dislike this kind of preprocessor trickery... but every rule has an exception.

The "pointer to function" data type and its use can be explained with a simple example. First, suppose we have a few functions such as:

```
void * foo(IntBag b) {printf("Hi\n");}
void * bar(IntBag b) {printf("Bye\n");}
```

The above code simply declares two functions that correspond to the type of function data type `_method` can point to.

Now consider the following code:

```
testMethod()
{
   _method f;  /* f is a pointer to a function */
   IntBag b = NULL;
   f = &foo;  /* set f to point to the "foo" function */
   (*f)(b);   /* LINE 6: same as foo(b); */
   f = &bar;  /* set f to point to the "bar" function */
   (*f)(b);   /* LINE 8: same as bar(b); */
}
```

In the above program, look at lines 6 and 8. On their own, the `(*f)(b)` syntax and previous declarations seem like an obscure and roundabout way to do something simple—invoke a function. Note, however, that both lines are *identical* but invoke *different* functions. Ultimately, it is this ability that make function pointers so powerful and worthwhile despite their ugly syntax.

Perhaps you can now guess why we use function pointers in the implementation of `IntBag`: this is how we can do different things dependent on the nature of the bag when, for example, we add an item to a bag.

Let us continue examining the private header file `IntBagP.h`. The next significant type definition is:

```
typedef struct _IntBag _IntBag, * _IntBagPtr;
struct _IntBag {
  void * data;
  union {
    void * ptr;
```

```
    int intVal;
  } state;
  _method *methods;
};
```

The three main components of the private `_IntBag` data structure are *data*, *state* and *methods*.

The *methods* field is declared as a pointer to a `_method` data type (and hence is a "pointer to a pointer to a function"); it will be allocated memory and used as the name of an array of function pointers.

The *data* field is used to store the data—i.e. the integers—in a concrete implementation. For example, the Vector implementation would use this field the same way the previous version used the field of the same name in the private structure `_IntVBag`. For the linked list implementation, this field would be used as the *head* field in the private `_IntLLBag` structure.

The *state* field is used to keep track of information relevant to the Bag as a whole. Note that this field is defined as a union containing either a integer or a generic pointer. The integer version is used when this is sufficient as is the case for the linked list implementation where only the size is maintained. For the vector implementation, two things are required: the size and the maximum size. In this case, the implementation will allocate a data structure for the two fields and set the *state* field of the `_IntBag` structure as a pointer to the implementation data structure.

The next part of the header defines constants that will be used to index into the array of methods:

```
enum {
  addMethod = 0,
  getSizeMethod,
  removeMethod,
  destroyMethod
};
```

As we have done before, we first show the general structure of the private implementation in Figure 5.3.

We now look at the implementation of `IntBag.c`.

The implementor in this case has chosen to make the linked list form of a bag the default. This is done with the following code:

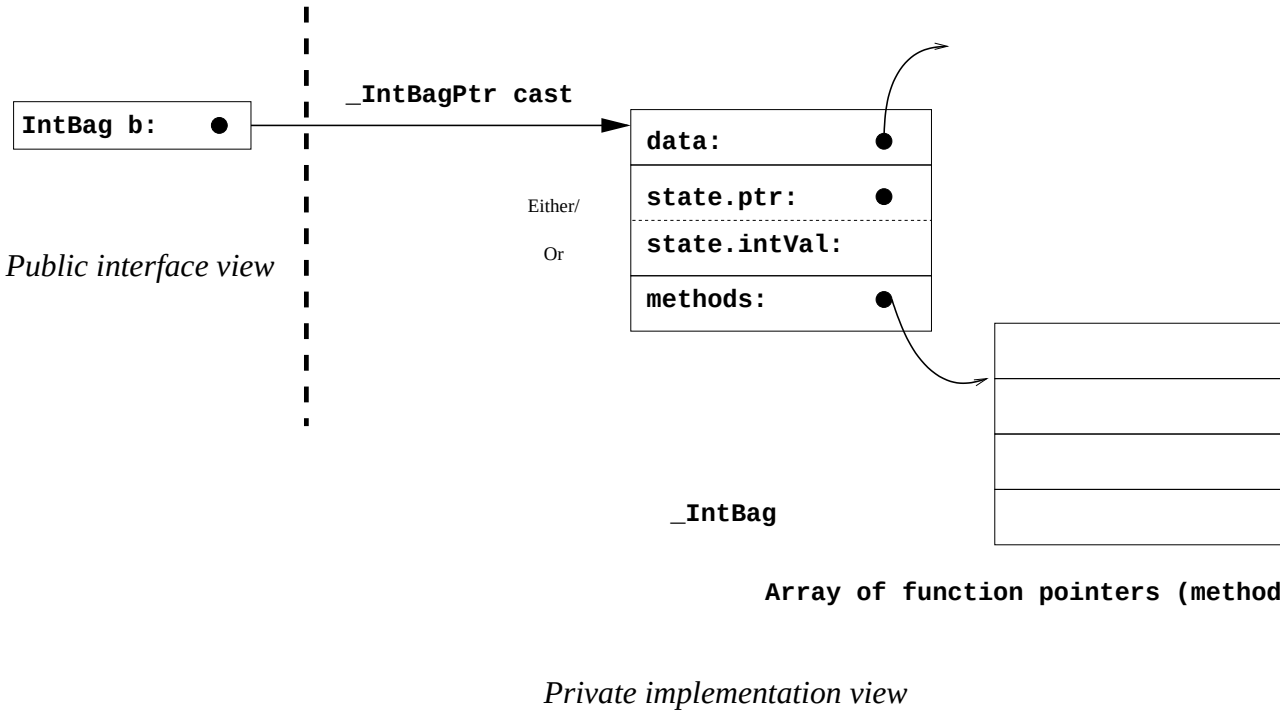


Figure 5.3: Basic Data Structures for an IntBag

```
#include "IntBagP.h"

public IntBag newIntBag(void)
 {
  return (IntBag) newIntLLBag();
}
```

The other functions are implemented by invoking the "real" method using the pointers in the methods function pointer array. For example, the `addIntBag` function is written as:

```
void addIntBag(IntBag b, int i)
{
  _IntBagPtr _b;

  _b = (_IntBagPtr) b;
  _b->methods[addMethod](b, i);
  return;
}
```

The other functions are similar:

```
int removeIntBag(IntBag b)
{
  _IntBagPtr _b;

  _b = (_IntBagPtr) b;
  return (int) _b->methods[removeMethod](b);
}

unsigned int getSizeIntBag(IntBag b)
{
  _IntBagPtr _b;

  _b = (_IntBagPtr) b;
  return (unsigned int) _b->methods[getSizeMethod](b);
}
```

```
void destroyIntBag(IntBag b)
{
  _IntBagPtr _b;

  _b = (_IntBagPtr) b;
  (void) _b->methods[destroyMethod](b);
  return;
}
```

Finally, let's look at how the linked list implementation of a integer bag is modified to correspond to the conventions required by `IntBag`. (The source code file is `IntLLBag2.c`.)

```
#include "IntBagP.h"

/* The _IntBag data structure is interpreted as follows:
 * struct _IntBag {
 *   void * data;       --- the "head" of the list of ints
 *   union {
 *     void * ptr;      --- NOT USED
 *     int intVal;      --- the "size" of the list
 *   } state;
 *   _method *methods;  --- the array of functions
 *                          operating on the list
 * };
*/

typedef struct _LList _LList, *_LListPtr;
struct _LList {
  int data;
   _LListPtr next;
};

private void _addIntLLBag(IntBag b, int i);
private int _removeIntLLBag(IntBag b);
private int _getSizeIntLLBag(IntBag b);

private _method theseMethods[] = {
```

```
  (_method) &_addIntLLBag,
  (_method) &_getSizeIntLLBag,
  (_method) &_removeIntLLBag
};

public IntBag newIntLLBag(void)
{
  _IntBagPtr b;
  b = malloc(sizeof(_IntBag));
  if (b == NULL)
    return NULL;
  b->data =  NULL;    /* b->data IS head of list */
  b->state.intVal = 0;      /* b->state.intVal IS size */
  b->methods = theseMethods;
  return (IntBag) b;
}

private void _addIntLLBag(IntBag b, int i)
{
  _LListPtr item;
  _IntBagPtr _b = (_IntBagPtr) b;

  item = malloc(sizeof(_LList));
  item->data = i;
  item->next = _b->data;
   _b->state.intVal++;    /* _b->state.intVal IS size */
  _b->data = item;        /*_b->data IS head of list */
  return;
}

private int _removeIntLLBag(IntBag b)
{
  int r;
  _LListPtr item;
  _IntBagPtr _b = (_IntBagPtr) b;

  if (_b->state.intVal <= 0) { /* _b->state.intVal IS size */
    eprintf("Fatal error, removing from empty bag\n");
```

```
  }
  _b->state.intVal--; /* b->state.intVal IS size */

  item = _b->data;    /* _b->data IS head of list */
  r = item->data;
  _b->data = item->next;  /* _b->data IS head of list */
  free(item);
  return r;
}

private int _getSizeIntLLBag(IntBag b)
{
  _IntBagPtr _b = (_IntBagPtr) b;

  return _b->state.intVal;  /* _b->state.intVal IS size */
}
```

Note that all of the functions except for `newIntLLBag` are private and hence inaccessible (by name) outside of this source code file. However, since `newIntLLBag` initializes the `methods` array to the addresses of the proper functions, they can all be invoked through these function pointers.

5.5 A generic Bag

This section is under construction; not part of ELE 428 in Winter 2000.

A simple way to create a Bag that can contain any kind of data is to speicify that the "data" is an opaque data type (i.e. a `void *` object).

To make things a bit more interesting, however, we will also extend the operations supported by a Bag to allow individual elements to be examined and modified and to add or remove an element anywhere in the Bag.

We can uniquely identify each element in the bag by associating an integer with each one. Although the Bag need not be implemented as an array, the user of the API can "think" of it an array (of unlimited size) with elements number from 0 to $n-1$ when there are n objects in the Bag.

5.6 Generic objects

This section is under construction; not part of ELE 428 in Winter 2000.

5.7 Remarks and Caveats

This section is under construction; not part of ELE 428 in Winter 2000.

5.8 Postscript: Using Java

This section is under construction; not part of ELE 428 in Winter 2000.

We stated previously that abstract data types are easier to implement in an object-oriented language. In this section, we breifly look at how we could do this in Java.

This section is meant for both readers who have some knowledge of Java and those who do not. For readers with no previous exposure to Java, some of the basic principles of the language are outlined and you should be able to follow the examples. Readers who know Java may also gain a better insight into the role of interfaces and abstract data types as well as the possible extensions to Java that would include generic objects.

The Java implementation of the kind of bags we have discussed in this chapter is so much easier that it is tempting to simply write the code for the most general kind of Bag directly.

5.9 Problems

5.1 How many different sets can be made if the only two allowed elements are the integers 1 and 2? Show all possible sets. How many different bags can be made with the same allowed elements?

5.2 The text suggests a different implementation of integer bags using an array whose size would be fixed at creation time. Discuss the possible merits and pitfalls of this approach and outline an implementation.

5.3 Thge text explains how the initial linked list implementation of an integer bag (`IntLLBag.c`) was converted into a general format (using pointers

to functions) compatible with `IntBag`. The new implementation is in source code file `IntLLBag2.c`. Convert `IntVBag.c` to `IntVGag2.c` that is compatible with `IntBag`.

5.4 Suppose that a C compiler did not include `realloc()` in its library but did have `malloc()` and `free()`. Implement a functional equivalent of `realloc()` (within the context of the `IntVBag` implementation where you have access to the `maxSize` field) using only `malloc()` and `free()`. Comment on how this version is likely to compare in efficiency with the real `realloc()`.

5.5 The text states (page 110):

> However, freeing the memory for the private data structure should only be done by the ADT implementation code.

Alec Smart chooses to ignore this advice. He reasons, "I *know* that the ADT data type is a void pointer and I also *know* that it is really a pointer to some memory chunk that was allocated dynamically. So I can free it myself instead of incurring the overhead of a `destroy` function call."

What do you think of Mr. Smart's reasoning?

5.6 The definition of the operations for a Bag state that the "getSize" operation is only a "convenience".

How would an application programmer determine the size of an IntLLBag in such a way that the bag itself remained unchanged and assuming that there was a function to determine if the bag were empty? Implement `myGetSize(IntLLBag b)`.

5.7 Consider another kind of bag where the order of removal is defined in terms of sorted order. In the case of integers, for example, the "remove" operation would return the smallest intger in the bag.

Outline at least two different ways such an ADT could be implemented. Commment on the $\Theta()$ complexity of the "add" and "remove" operations for each case.

5.8 Suppose we wanted to implement Sets instead of Bags. (A Bag can contain duplicates, a Set cannot). Discuss the API interface and the implementation using a linked list (`IntLLSet`). Give the $\Theta()$ complexities for the add and remove operations as a function of set size n.

5.9 For the linked list implementation, items are added and removed at the beginning of the list. Suppose they were added and removed at opposite ends. What whould the $\Theta()$ complexities of each of the operations be?

Part II

Data Structures

Version 1.0 April 8, 2004

Chapter 6

Stacks and Queues

In this chapter we examine the interfaces and implementation of three common data structures: *stacks*, *queues* and *priority queues*. All three of these data structures are *bags* of the type discussed in Chapter 5. However, unlike a generic bag which is an unordered collection of objects with no defined removal order, the stack, queue and priority queue structures do define the order.

For a stack, elements are removed in the opposite order that they were added (this is also called a *Last In First Out* or *LIFO* data structure); for a queue, elements are removed in the same order they were added (also called a *First In First Out* or *FIFO* data structure); and, for a priority queue, elements are removed according to their "priority" with the highest priority item being removed.

For example, if the integers 2, 5, and 3 are added to a stack, a queue and a priority queue in that order, then they will be removed in the order:

stack: 3, 5, 2;

queue: 2, 5, 3;

priority queue: 5, 3, 2 (assuming that the item's priority is its integer value).

These "data structures" correspond to situations in everyday life. A queue is usually how people organize themselves when waiting in line for an ATM machine or some other service. In this case, the person who has waited the longest is the next one serviced.

On the other hand, the order that patients are treated in the emergency room of a hospital is quite different. If one patient needs treatment for a hang nail and another arrives carrying some important bodily part that had been cut off, it is reasonable that the person with the significant missing "major part" (not a finger nail) would be treated before the patient with the hang nail problem. Typically, a tirage is done to sort patients in order of the seriousness of their injury and the "time of arrival"—i.e. the order they were added to the queue—is *not* the primary consideration on when they are treated.

A stack is commonly encountered when you place things on top of each other to create a pile. Obviously, the first thing you put down is at the bottom of the pile and the last thing you placed is at the top. When you remove things from the pile, you are well advised to start at the top (to avoid having the pile crash down).

6.1 Stacks

The stack is one of the most widely used data structures in computer engineering. Indeed, it is so important that almost all modern central processing units (CPUs) implement one or more stacks directly in hardware.

Like any Bag, a stack supports the add and remove operations. However, "add" is usually called the "push" operation and "remove" the "pop" (or "pull") operation.

6.1.1 The uses of stacks

Reversing the order of items: The central characteristic of a stack is that it reverses the order of items that were pushed onto it when they are popped.

The algorithm for using a stack to reverse the order of things is straight forward:

ReverseWithStack Algorithm

Output the reverse of n *objects (the input)*

Step 1: Create an empty stack.

Step 2: Push each object in turn onto the stack.

Step 3: Pop each object in turn off the stack until the stack is empty. (The will be popped in reverse order.)

Step 4: STOP.

Converting recursive algorithms to iterative ones: We have already seen on page 27 the general form of many divide-and-conquer recursive algorithms and have seen that some of these algorithms (tail recursive, see page 31) can be mechanically converted to iterative implementations. We now examine how some other recursive algorithms that are not tail-recursive (such as towers of Hanoi or count change) can be converted to iterative algorithms with the use of a stack.

RecursiveToIterative Algorithm

Convert a recursive algorithm to iterative form using a stack

Step 1: Create an empty stack.

Step 2: Push the parameters onto the stack.

Step 3: If the stack is empty, STOP.

Step 4: Pop the parameters off the stack.

Step 5: If the problem is simple enough to solve directly, solve it and, if necessary, combine this solution to the total solution. Go back to *Step* 3.

Step 6: Otherwise, split the problem into one or more simpler problems.

Step 7: Push the parameters for each of these simpler problems onto the stack.

Step 8: Go back to *Step 3*.

Balancing: Many formal languages require that certain kinds of tokens be balanced. For example, arithmetic expressions that allow parentheses must be written so that there is a balancing right parenthesis for each

left parenthesis. Thus the expression "`(1+2*(1+1))`" is legal, whereas "`(1+2))*2(`" is illegal even though there are the same number of left and right parentheses or any kind of balanceable token that has both left and right versions.

A simple algorithm using a stack to verify that left and right parentheses balance is:

BalanceLeftRight Algorithm

Verify that left and right tokens balance

Step 1: Create an empty stack.

Step 2: If there are no more tokens: if the stack is empty the input is balanced; otherwise it is not balanced. STOP.

Step 3: If it is a left token, push it onto the stack and go back to *Step 2.*

Step 4: Otherwise, pop the stack. If the token type is not the same as the right token, the input is not balanced and STOP. Otherwise, go back to *Step 2.*

Subroutine linkage: Many CPUs implement a stack in hardware to aid in subroutine linkage and other common operations. At the machine language level, a CPU maintains a register called the Program Counter or Instruction Pointer (PC) containing the address of the next machine instruction to execute. Most instructions simply perform their operations and increment the PC to the next instruction in sequence. But some instructions modify the default sequence. There are both conditional and unconditional "goto" instructions which directly modify the value of the PC.

Our concern here is with the "goto subroutine" (often called `jsr`) and "return from subroutine" (`rts`) instructions. The operations performed by these instructions are:

jsr: 1. Push the value of the PC onto the hardware maintained CPU stack. (Note the the PC will contain the address of the instruction that immediately follows the `jsr`.)

2. Modify the PC to the argument specified in the `jsr` instruction.

rts Pop the top of the hardware stack into the PC.

Consider, for example, the following outline of function calls:

```
main()
{
   foo();
}

foo()
{
   bar();
   ...
   bar();
}

bar()
{
   ...
}
```

The compiler would generate machine language instructions like:

```
main:
     .
     .
   jsr foo
   instruction A
     .
     .
foo:
     .
     .
    jsr bar
    instruction B
```

```
        .
        .
    jsr bar
    instruction C
        .
        .
    rts

bar:
        .
        .
    rts
```

Initially, the PC contains the address of the first instruction of the "main" routine and execution begins there with an empty[1] stack. When the instruction `jsr foo` instruction is executed, the address of the next instruction—A—is pushed onto the stack and the machine then executes the first instruction of the `foo` routine. Eventually, the first `jsr bar` instruction is encountered. The address of the next instruction—B—is then pushed onto the stack and control is transferred to the first instruction of the `bar` routine. At this point the top of the stack is the address of `Instruction B` and below it is the address of `Instruction A`.

The `bar` routine then is executed and the last instruction performed is the `rts` instruction at the end. This pops the top of the stack, which contains the address of Instruction B and sets the PC to this address. Hence the next instruction executed is Instruction B as desired. Note also that the stack is now in the same state it was prior to the execution of the `jsr bar` instruction—i.e. the top of the stack is the address of Instruction A.

Later in the `foo` routine, another `jsr bar` instruction is executed. This time the address of Instruction C is pushed onto the stack (which now contains C, A) and the `bar` routine is executed once again. As before, the `bar` routine finishes by executing the `rts` instruction. This time, however, the top of the stack contains the address of Instruction C which is popped and placed in the PC. Thus execution continues at Instruction C in the `foo` routine and

[1]Usually the stack is not empty; its contents are defined by the *loader/linker*, the command interpreter and the operating system. If it were really empty, returning from `main` would cause unpredictable behavior.

the stack again contains only the address of Instruction A. Finally, the `rts` instruction at the end of `foo` is executed and the address of Instruction A in the main routine is popped into the PC. Execution then continues in the main routine.

6.1.2 The implementation of stacks

An abstract data type corresponding to the removal order defined by a stack is not difficult to implement. Indeed, the linked list and vector implementations of `IntBag` do in fact remove their contents the same way a stack should. Let us repeat once again, however, that this behavior cannot be relied on in general.

If one did want to use the implementation of `IntLLBag` as an `IntLLStack`, copy the source code files into similarly named files substituting "Stack" for "Bag". Similarly, edit the files and make the same substitution. Finally, and most importantly, write new interface API documentation for the abstract stack data type.

It often occurs, however, that the overhead of an ADT stack implementation is not required. It is common, for example, for an algorithm to require only a single stack containing a well-defined data type and whose maximum possible size is predictable at the outset. In such cases, a simple array implementation of a stack may be appropriate.

The API that the stack should implement a single stack of some arbitrary data type "DataType" is:

void push(DataType item): Pushes `item` onto the stack.

DataType pop(void): Pops and returns the item on top of the stack. The behavior is undefined if the stack is empty.

int isEmptyStack(void): Returns non-zero (true) if the stack is empty; returns 0 if there are items on the stack.

void initStack(void): Initializes an empty stack.

A simple generic implementation of this API is shown below:

```
typedef int StackData;   /* For example */
static unsigned int top = 0;
```

```
#define STACK_SIZE 1000
static StackData stack[STACK_SIZE];

static void push(StackData p)
{
  stack[top++] = p;
  return;
}

static void initStack()
{
  top = 0;
  return;
}

static StackData pop()
{
  return stack[--top];
}

static int isEmptyStack()
{
  return (top == 0);
}
```

6.1.3 Examples

reverse The function `reverse(int data[], int n)` reverses the array of integers `data`; the argument n gives the size of the array.

The implementation simply uses the ReverseWithStack algorithms described on page 6.1.1.

```
void  reverse(int d[], int n)
{
  int i;

  initStack();  /* Create an empty stack */
```

```
  /* Push each object in turn onto the stack. */
  for(i = 0; i < n; i++) {
    push(d[i]);
  }
  /* Pop each object in turn off the stack
   *  until the stack is  empty.
   */
  for(i = 0; i < n; i++) {
    d[i] = pop();
  }

  return;
}
```

The stack used in the `reverse` function is the same generic one shown previously. (The complete source code and a main driver routine can be found in Appendix E or in the file `src/dataStructsAndPtrs/reverse.c`.)

Iterative version of Towers of Hanoi:

We can use the RecursiveToIterative algorithm (page 133) to convert the original recursive version of the Towers of Hanoi algorithm (page 38) to an iterative implementation using a stack.

The algorithm uses a single stack that contains the parameters to the `towers` function. We first define a data type corresponding to this kind of stack and set up the constants, static variables and array used for implementing the stack.

```
typedef struct
{
  int n;
  int from;
  int to;
}  Params, StackData;

static unsigned int top = 0;
#define STACK_SIZE 1000
```

```
static StackData stack[STACK_SIZE];
/* standard implementation of stack not shown */
```

Note that we have given the data structure two names: `Params` and `StackData`. These two type names are aliases of each other. We use `StackData` as the data type name so that we can just copy our generic single stack implementation directly into the source code file. We use the other name—`Params`—in the `towers` function since this name more closely corresponds to the problem domain and the implementation of the algorithm.

Before converting the algorithm to iterative form, we make a slight modification to the original recursive implementation as follows:

```
void towers(int n, int from, int to)
{
  if( n == 1)
    printf("%d %d\n", from, to);
  else {
    int spare = 6 - from - to;
    --n;
    towers(n, from, spare);
    towers(1, from, to);
    towers(n, spare, to);
  }
}
```

In the original version, the `printf` statement was used in place of the recursive call `towers(1, from, to)`. This recursive version works just like the original one, but the explicit special treatment for moving a single disk makes the conversion to an iterative solution easier. (One of the problems and its answer explores this in greater detail.)

We now apply the RecursiveToIterative algorithm to the recursive function shown above.

First, we create an empty stack and push the parameters onto it (*Steps* 1 and 2 of the RecursiveToIterative algorithm):

```
void towers(int n, int from, int to)
{
  Params p;
  initStack();
  p.n = n; p.from = from; p.to = to;
  push(p);
```

The rest of the algorithm is basically a loop that continues until the stack is empty. Inside the loop, we pop parameters off the stack and solve base case problems if we can (i.e. if there is only one disk to move). Otherwise, we divide the problem into three simpler subproblems and push the parameters to each onto the stack. We have to be careful, however, in the order we solve the sub-problems. The order in the recursive version is:

```
towers(n, from, spare);
towers(1, from, to);
towers(n, spare, to);
```

If we pushed `towers(n, from, spare)` first, it would be solved last, but we want it solved first. Hence we have to push the arguments for each of the sub-problems in the opposite order that the recusive calls are made. The loop is:

```
  while(!isEmptyStack()) {
    p = pop();
    n = p.n; to = p.to; from = p.from;
    if (n == 1)
      printf("%d %d\n", from, to);
    else {
      int spare = 6 - from - to;
      --n;
      p.n = n; p.from = spare; p.to = to;
      push(p);
      p.n = 1; p.from = from; p.to = to;
      push(p);
      p.n = n; p.from = from; p.to = spare;
```

```
      push(p);
    }
  } /* END-LOOP while(!isEmptyStack()) { */
}
```

HTML checker: The HTML (Hypertext Markup Language) requires that tags be balanced. A tag is defined with the following BNF:

```
<tag> ::= <startTag> | <endTag>
<startTag> ::= '</' <name> '>'
<endTag> ::= '<' <name> '>'
<name> ::= a sequence of alphabetic characters
```

For example, the following skeletal HTML fragments are grammatically correct:

```
<h3> <a><b> </b> </a></h3>
<foo> </foo> <bar> </bar>
```

The following are incorrect:

```
<h3> <a><b> </a></b></h3>
<foo> </foo> </bar> <bar>
```

The `token` data type is defined as:

```
typedef enum {TStart, TEnd, TEOF} TagType;
typedef struct {
   char * tag;
   TagType type;
} Tag;
```

The pseudocode for the tokenizer is:

```
Skip to '<' or EOF
if (next char == '/')
   collect the following characters until '>'
   set token type to TStart and return
else
   collect the chars until '>'
   set token type to TEnd and return
```

The implementation of a function to verify that the tags are balanced follows directly from the BalanceLeftRight algorithm and the types defined above. The function is called `isBalancedHTML` and returns true or false depending on whether the input is balanced or not. The C implementation is:

```
enum {false=0, true=1};

int isBalancedHTML(void)
{
  Params p;
  char * leftTag;
  initStack();
  while ((token = getNextToken()).type != TEOF) {
    if (token.type == TStart) {
      push(token.tag);
    } else {
      leftTag = pop();
      if (strcmp(leftTag, token.tag) != 0) {
        return false;
      }
    }
  }
  if (isEmptyStack())
    return true;
  else
    return false;
}
```

6.1.4 "Peekable stacks"

A variation on the pure stack that allows only the push and pop operations is a "peekable" stack. As with an ordinary stack, only push and pop operations can change the size of the stack. Unlike a regular stack, however, it is possible to examine or modify any item on the stack.

In short, the API for a peekable stack has the same operations as a regular stack:

void push(DataType item): Pushes `item` onto the stack.

DataType pop(void): Pops and returns the item on top of the stack. The behavior is undefined if the stack is empty.

int isEmptyStack(void): Returns non-zero (true) if the the stack is empty; returns 0 if there are items on the stack.

void initStack(void): Initializes an empty stack.

The API is augmented with three additional operations to get or set items on the stack and determine the total number of items on the stack. Stack items are referred to by their *index*. We assume that the top of the stack is, by definition, at *index*=0. The item under it would have an index of 1 and so on. A negative index or an index greater than or equal to the total number of items on the stack would be illegal.

DataType set(int index, DataType data): Set the item on the stack *index* elements from the top to the data. It returns the old value that is replaced. The number of items on the stack is not modified. The behavior is undefined if *index* is not in the valid range 0–*Stack size*.

unsigned int getSize(void): Returns the number of items in the stack.

DataType get(int index): Returns the value of the item *index* elements from the top. For example, when used as `get(0)` the value of the item on top of the stack is retrieved, but it is *not* popped from the stack. The behavior is undefined if *index* is not in the valid range 0–*Stack size*.

The peekable stack is easy to implement when there is only a single stack of known finite size. A statically allocated array can then be used for the stack as seen previously.

6.1.5 Stack Frames

The peekable stack has an important application in the implementation of programming languages like C that allow recursion, the passing of parameters to functions and the declaration of variables local to a function. As in the case of using a stack for subroutine linkage, we will assume here that the stack considered here is a hardware stack directly supported by the CPU architecture. The address of the top of these stacks is maintained in a hardware register and the CPU architecture allows an "indexed addressing mode" whereby items anywhere in the stack can be examined and modified. In particular, local variables[2] and passed parameters are placed on the stack. And, as we have already seen, the stack is used to store the return address that a function should return control to when it has finished its work.

The most common protocol to do this in C language implementations requires that the function that invokes (the *caller*) another function (the *callee*) respect the following conventions.

Caller conventions: To invoke a function (including a recursive call to itself), the calling function does:

1. Push the parameters onto the stack. (In C, the parameters are pushed onto the stack in the opposite order to their declaration. Thus the top of the stack at the end of this step is the first parameter to the function being called.)
2. The return address is pushed onto the stack.
3. Transfer control to the called function.
4. When control is transferred back (at which point the stack will still have the parameters on it, but not the return address), pop the parameters off the stack.

Callee conventions:

1. The called function pushes space for its local variables onto the stack.
2. In implementing the function, all references to local variables or passed parameters are performed using a "peekable" stack interface.

[2]Except for *static* local variables which, although they have the same scope as ordinary locals, are placed at a globally allocated constant location in memory

3. When the function has finished its work, it deallocates the space used on the stack for local variables. (Conceptually, it pops them but ignores their values since it has finished with them.)
4. The top of the stack is now the return address that control should be transferred to. This address is popped and placed in the Program Counter as before.

```
void bar(int x, int y)
{
   int p, q;

   /* do stuff*/
   return;
}

void foo()
{
    bar(5, 6);
}
```

Figure 6.1 shows what the stack would look like after `bar` had been invoked and had allocated space for its local variables p and q.

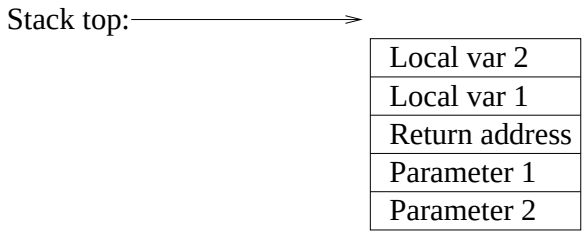


Figure 6.1: Example of a Stack Frame (without a Frame Pointer)

6.2 Queues

The most common kind of queue in everyday life is the "First-In, First-Out" (FIFO) queue often (frustratingly) encountered when waiting for some service. This is what we call a queue.

The API for a queue is almost identical with a Bag API except that the order of removal is defined. The most abstract description of the required operations on a queue are:

add Add something to a queue.

remove Remove the item that has been in the queue the longest time.

The more detailed API for ADT queue that could contain an unlimited number of objects is:

Queue newQueue(void): Returns a newly created *Queue* or `NULL` if one cannot be created.

void addQueue(Queue q, Object obj): Adds *obj* to the specified queue *q*.

Object removeQueue(Queue q): Removes an object (and returns it) from the specified queue *q*. The program exits if the queue is empty. Note that the order of removal is *not* specified.

unsigned int getSizeQueue(Queue q): Returns the number of items in queue *q*.

void destroyQueue(Queue q): Destroys a previously created Queue, releasing all its resources.

6.2.1 Implementation

The basic rule to remember when implementing a queue can be summarized as "Add at rear, remove at front". (Note that the API does *not* specify how things are added to a queue, only how they are removed. Hence the implementor is under no *obligation* to respect the "add at rear, remove at front" aphorism, but it is the simplest way to implement a queue.)

Because we access both "ends" of the queue, a doubly-linked list offers straight forward implementation which we outline below:

```
typedef struct Queue Queue, Node, *NodePtr;
struct Queue {
   DataType info;
```

```
   NodePtr next;
   NodePtr prev;
}
static NodePtr head = NULL, tail = NULL;

void addQ(DataType item)
{
   NodePtr n;
   n = malloc(sizeof Node);
   n->info = item;
   if (head == NULL) {
      head = tail = n;
      n->next = n->prev = NULL;
   } else {
      NodePtr p = tail->prev;
      if (p == NULL) {
          n->prev = tail;
      } else {
          n->prev = p;
      }
      n->next = NULL;
      tail = n;
   }
}

DataType removeQ(void)
{
    NodePtr gone = head;
    DataType item = gone->info;
    head = gone->next;
    free(gone);
    return item;
}
```

(The implementation of a doubly-linked list can be simplified by using a circular list.)

A single queue of known maximum size can also be implemented using an array of fixed dimension.

We maintain two indices that keep track of where the *front* and *rear* elements are in the array. These indices are called, unsurprisingly, `front` and `rear`. We will stipulate, somewhat arbitrarily, that `front` will be the index to the front element (the one that has been there the longest) in the queue. The `rear` index will indicate where the next item will be added to the queue.

Ignoring, for the moment, the finite size of the array, we can add and remove elements from the queue as follows:

```
typedef int DataType;   /* for example */
enum {QUE_MAX=1000, QUE_ARR_SZ=QUE_MAX+1};    /* for example */
static DataType queue[QUE_ARR_SZ];

/* incorrect skeleton---assumes infinte array size */
static DataType remove(void)
{
   return queue[front++];
}

/* incorrect skeleton---assumes infinte array size */
static void add(DataType item)
{
   queue[rear++] = item;
}

static int getSize(void)
{
   return rear - front;
}
```

The basic idea expressed above is correct, but we have not accounted for the finite size of the array. As we add more things to the queue, `rear` will eventually be incremented beyond the end of the array.

This is illustrated in Figure 6.2.

The solution is to reset `rear` to the beginning of the array when it would "fall off the end". We do the same thing for `front`. Hence, the implementation becomes:

```
typedef int DataType;   /* for example */
```

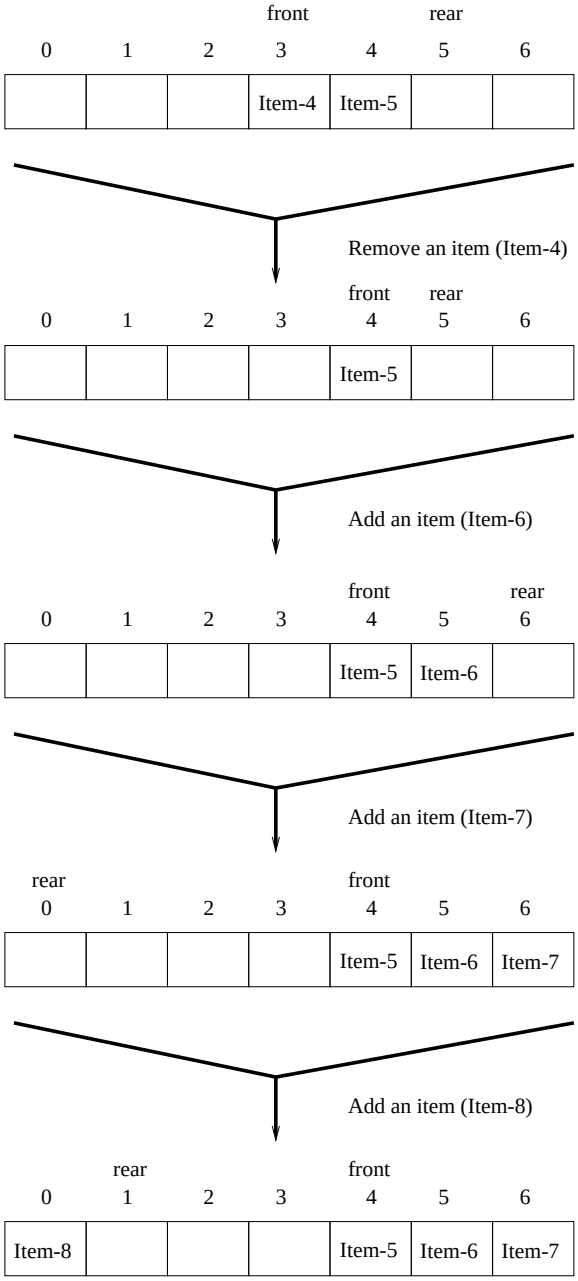


Figure 6.2: Queue implemented as a "circular" array

```
enum {QUE_MAX=1000, QUE_ARR_SZ=QUE_MAX+1};     /* for example */
static DataType queue[QUE_ARR_SZ];

static DataType remove(void)
{
   DataType item;

   item = queue[front++];
   if (front >=  QUE_ARR_SZ)
       front = 0;
   return item;
}

static void add(DataType item)
{
   queue[rear++] = item;
   if (rear >= QUE_ARR_SZ)
       rear = 0;
}

static int getSize(void)
{
   int size;
   size = rear - front;
   if (size < 0)
       size +=  QUE_ARR_SZ;
   return size;
}
```

6.3 Priority Queues

Like the ordinary queue, a Priority Queue differs from a Bag in that the order of removal is specified. In addition, however, there must be some information in items added to a Priority Queue that allows their importance (i.e. priority) to be compared.

We will not discuss priority queues in detail here because one of the most elegant ways to implement them in the most general case and that

allows $\Theta(\log n)$ behavior for both adding and removing will be explored in Chapter 7.

Nonetheless, there are simple implementations that are good enough in some circumstances.

Simple Bag and Sort: A simple way to implement a priority queue is to use an unordered Bag (such as a simple array) for adding elements and sorting the whole thing to find the highest priority item for removal. Adding an item would be a $\Theta(1)$ operation. Since sorting can be done in $\Theta(n \lg n)$ time, removal from this kind of priority queuue would be $\Theta(n \lg n)$. In some situations, this perfromance penalty might be acceptable.

Sorted linked list: Another simple way to implement a priority queue is to maintain a sorted bag (such as a sorted linked list). If it is a linear structure, then both adding and removing would be $\Theta(n)$ in the worst case.

6.3.1 Delta time queue

6.4 Problems

6.1 Re-write the *CountChange* program without using recursion.

6.2 An expression that uses the parenthetical balancing characters`(){}[]` is checked using using the BalanceLeftRight algorithm. For each of the following expressions, indicate whether the algorithm will determine if it is correctly balanced or not. If it is correctly balanced, show the stack contents after the seventh token. If is not balanced, show the stack just before the "pop" operation that detected the error.

1. `([{(}])}])`
2. `([{(})}}])`
3. `([{(}])]})`

6.3 Define a template implementation of a peekable stack. Assume that only a single such stack is used in a module and that its maximum size can be determined at compile time. (Use the example of how a single stack with compile-defined maximum size was implemented as a set of static functions.)

6.4 Implement an abstract priority queue for integers using a sorted doubly-linked list implementation. Call the implementation `IntDLLPriQ`.

6.5 Some HTML tags do not need to be balanced (although they can be). The paragraph tag `<p>` is an example.

How would you modify the BalanceLeftRight algorithm so that it would not indicate an error in these cases?

(For example, the following sequences would be considered valid:

```
<foo> <p> </p>  </foo>
<foo> <p>  </foo>
<foo> <p> <p> </p>  </foo>
```

but these would not be:

```
<foo> <p> </foo> </p>
<foo> <p>  <bar> </foo>
<foo> </p>  </foo>
```

)

Modify the C program for these cases.

6.6 Consider the following skeletal C code:

```
main() {
main_1:   foo(5, 6);
main_2:
}

foo(int p, int q)
{
foo_1:  int x, y, z;
foo_2:  x = p;
}
```

Show what the stack frame looks like just before executing the line labelled `foo_2`.

6.7 The code to remove an item from a Queue implemented as a linked list was written as:

```
DataType removeQ(void)
{
    NodePtr gone = head;
    DataType item = gone->info;
    head = gone->next;
    free(gone);
    return item;
}
```

Why not write it as:

```
DataType removeQ(void)
{
    NodePtr gone = head;
    head = gone->next;
    free(gone);
    return gone->next;
}
```

6.8 Implement a Queue using a doubly-linked circular list.

6.9 Suupose a stack, queue and priority queue have each been implemented using an array of fixed size 5 and standard implementations. All three are initially empty. The items in the Priority Queue are ordered by their numeric values. A sequence of add and remove commands are issued for each data structure. For each of the operations, indicate whether an error (overflow or underflow) occurs; for remove operations that do not cause an error, what value is removed?

1. Sequence 1:
   (a) add 5
   (b) add 9
   (c) add 4
   (d) add 3
   (e) remove
   (f) remove

(g) remove

(h) remove

2. Sequence 2:

(a) add 5

(b) add 4

(c) remove

(d) add 3

(e) remove

(f) remove

(g) remove

3. Sequence 3:

(a) add 5

(b) add 4

(c) remove

(d) add 3

(e) add 6

(f) add 8

(g) add 2

(h) remove

(i) remove

(j) remove

6.10 Modify the simple array implementation of a Queue so that the program exits if an underflow or overflow error occurs.

6.11 Modify the simple array implementation of a Stack so that the program exits if an underflow or overflow error occurs.

6.12 What is the $\Theta()$ complexity of the simple array implementations of the add and remove operations for a Queue? ...for a Stack?

Chapter 7

Trees

> [*Trees are*] *the most important nonlinear structures that arise in computer algorithms.*
>
> —Donald Knuth

> *I think that I shall never see*
> *A poem as lovely as a tree.*
>
> —Joyce Kilmer

The tree data structure is often used to represent any kind of information that can be organized in a hierarchical way. We have already used trees informally when we illustrated concepts like recursion, recurrences and parsing with eponymous "recursion trees", "recurrence trees" and "parse trees".

In this chapter we describe how trees can be represented conceptually (especially using various visual methods) and formally, including their representation as data structures in programming languages. The basic concepts, terminology and algorithms commonly used with trees is also discussed.

Much of the chapter involves binary trees, a structure very closely related to a general tree.

7.1 What is a tree?

We begin with an intuitive description of various kinds of hierarchical information that can be represented as a tree.

A book: A book (such as the one you are reading) is divided into chapters, the chapters are subdivided into sections, and so on. The Table of Contents reflects this hierarchical structure. For example, a partial description of this book's structure is:

1 `Algorithms`

- **1.1** `What is an algorithm?`
- **1.2** `A simple sort algorithm`
- **1.3** `A better sort algorithm`
  - **1.3.1** `Merge`
  - **1.3.2** `The Merge Sort algorithm`
  - **1.3.3** `The analysis of ‘‘merge sort’’`
- **1.4** `Implementation`

2 `Recursion`

In this case, the deeper levels of the hierarchy are represented by greater indentation and with increasingly detailed numerical identifiers.

Pedigree trees: This kind of tree shows the ancestors of an individual. The first example (Figure 7.1) shows *your* (generic) pedigree.

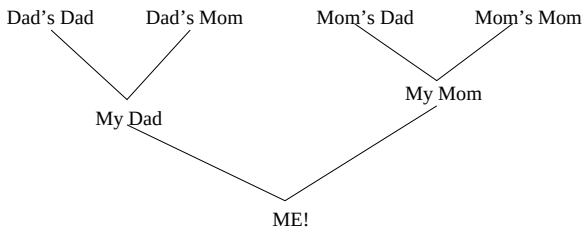


Figure 7.1: My ancestors

The second example illustrated in Figure 7.2 shows the pedigree of an (imaginary) person "Bill". Here, however, we have rotated the names of the parents (and grandparents, etc.) to achieve a different visual presentation.

Lineal family chart: The next example shows part of a family tree, but in the "opposite direction". In these charts, the descendants of an

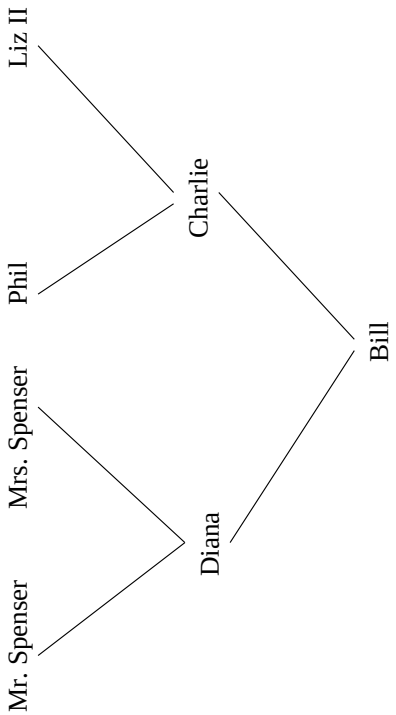


Figure 7.2: Bill's ancestors

individual are shown. Figure 7.3 shows some of Noah's descendants (according to the account in Genesis).

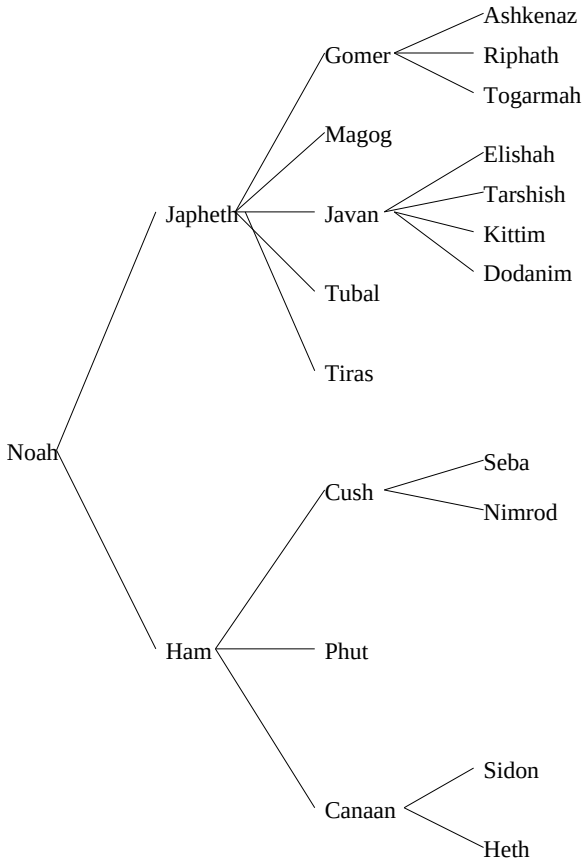


Figure 7.3: Noah's descendents

Organizational chart: Organizations including the military, companies, government agencies and so forth are usually organized (at least partly) in a hierarchical way. A typical organizational chart is shown in Figure 7.4.

Dependency diagram: Programs and projects are often organized as a collection of source code files that generate object and executable files. A *dependency diagram* shows how target files depend on other files. A typical example is shown in Figure 7.5.

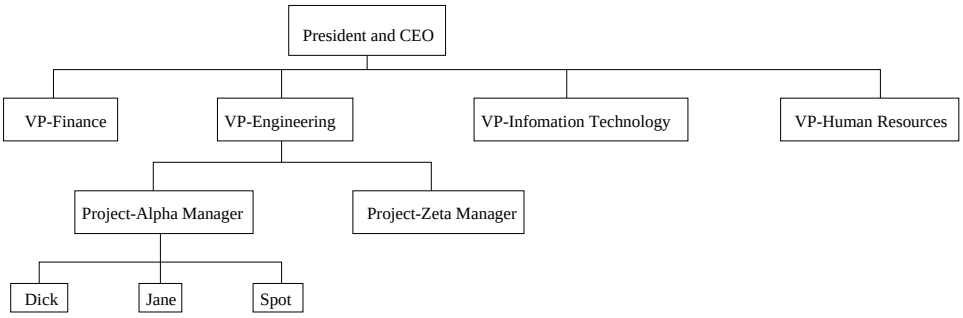


Figure 7.4: Organizational chart

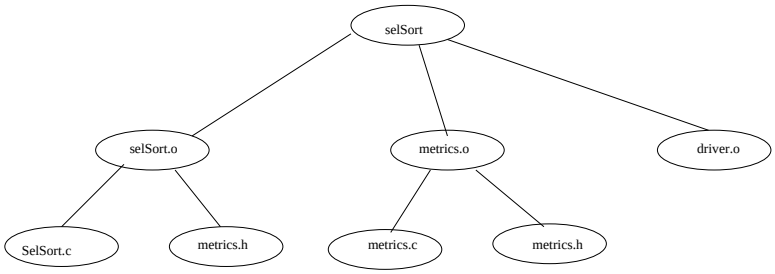


Figure 7.5: Dependency Chart

Expression tree: These trees (similar to parse trees) show how an expression is evaluated. A simple example is shown in Figure 7.6.

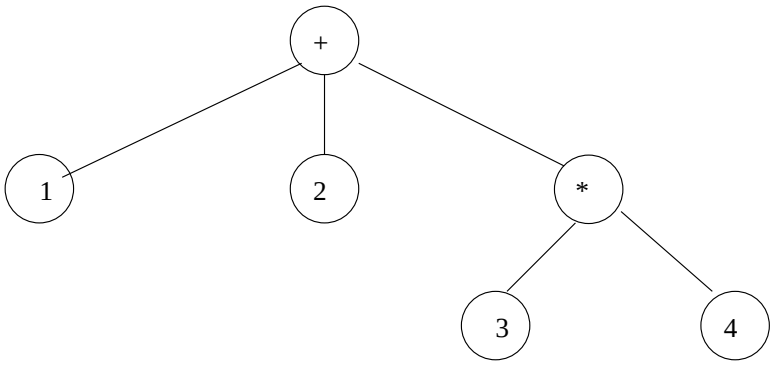


Figure 7.6: Expression tree

Remarks

In all cases, there is a single node that is at the highest hierarchical level. We call this node the *root* of the tree. Note that these trees are drawn in different ways. The root was placed at the left side (Noah's descendents), at the right (Bill's ancestors), at the bottom (my ancestors) or at the top (organizational chart). In the case of the book example, a figure was not drawn but the nodes were represented using typographical conventions that highlighted its tree-like structure.

The choice of which way to draw or represent a tree is arbitrary. You should use whichever visualization best corresponds to your conceptual understanding of how the information is organized. Despite this, we will adopt a standard or canonical representation of trees most of time. In this representation, the root is placed at the top and is represented by a circle or oval with textual information placed inside. Lower level nodes are placed below higher ones and connected by lines. An example of this kind of representation is shown on Figure 7.7.

It is also important to recognize that a tree structure does not always represent all the important information about relations and interactions in a real entity. For example, although the organizational chart of Figure 7.4 does correctly reflect the basic hierarchy, it does not show important relationships. Suppose that Spot believes he is being harassed by his manager

who, Spot believes, is just "throwing him some bones". The Human Resources department deals with this kind of harassment issue, but the tree organization—taken literally—would imply that Spot would have to communicate his complaint up through the hierarchy to the president who would then forward it down the HR chain to the appropriate officer. This is clearly absurd; in a real organization there would be other communication channels that would allow Spot to take his complaint directly to the HR officer.

It is also important to recognize that not all things called "trees" in everyday language correspond to the kinds of trees discussed here. For example, a traditional "family tree" is *not* a tree. (One of the problems—and its answer—explores this.)

7.2 Definitions and terms

We now give a formal definition of a tree.

A tree is defined as a collection of one or more *nodes* such that:

1. One of the nodes is designated as the *root*.

2. The remaining nodes (if any) are partitioned into collections, each of which is itself a *tree*. These trees (if any) are called the *sub-trees* of the *node*.

We often represent a tree as a diagram with the root at the top as shown in Figure 7.7.

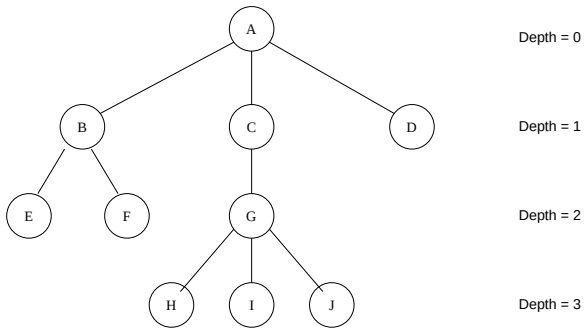


Figure 7.7: A simple tree

There are various terms related to trees that we will use with precise meanings as given here.

Child: The root of each subtree of a root is called a *child* of the root.

Parent: Each node has precisely one parent. The root's parent is NIL.

Edge (or Arc): A parent-child relationship. This is represented as a line connecting two nodes in our canonical diagram of a tree.

Sibling: All nodes with the same parent are *siblings*.

Ancestor: A node that is the parent, the parent's parent, on so on is an *ancestor* node.

Descendant: A node's children, their children and so on are descendant nodes.

Path: A set of edges connecting a node with a descendant node. A path always exists between a node and any of its descendants and the path is *unique*. The number of edges in the path is called the path's *length*.

Internal node: A node with at least one child.

External node (or leaf): A childless node.

Depth (of a node): The depth of the root is 0. The root's children are at depth 1; the root's children's children (*grandchildren*) are at depth 2 an so on.

Height (of a tree): The height of the tree is the maximum depth of any any node. An equivalent definition is the length of the longest path from the root to any node in the tree.

Ordered Tree: The basic definition of a tree gives no meaning to the order that children appear in the tree's representation. When the order does matter, we say it is an *ordered tree*. Except for the example of the book, all of the trees discussed informally in the previous section were unordered.

We can represent a tree using parenthetical notation. The BNF is:

```
<Tree> ::= '(' <root> { <Tree> } ')'
<root> ::= alphanumeric characters
```

Figure 7.8 shows some examples of some simple trees and their corresponding parenthetical representation.

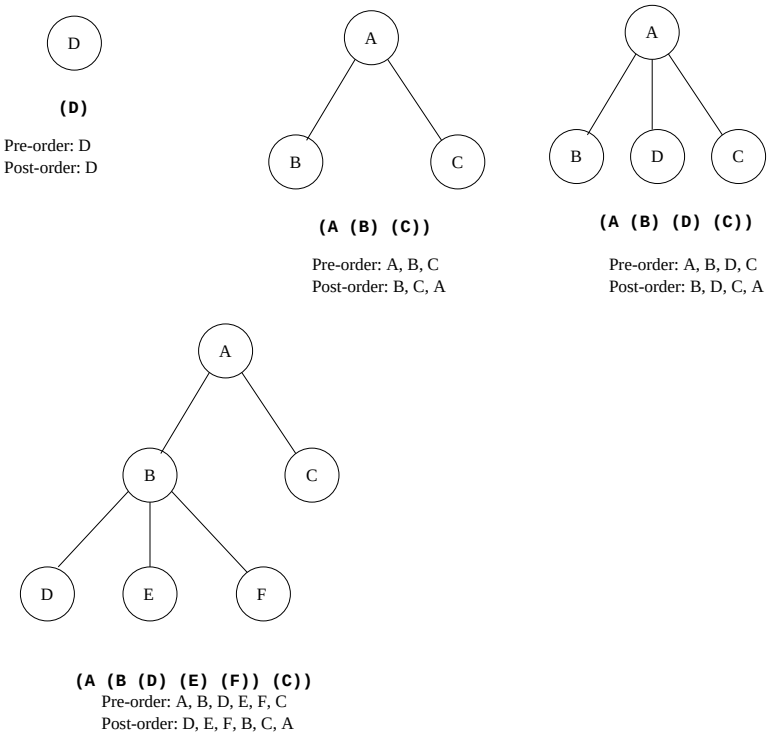


Figure 7.8: Examples of Trees with parenthetical representation

For example, the tree shown in Figure 7.7 can be represented as:

```
(A (B (E) (F)) (C (G (H) (I) (J))) (D))
```

7.3 Representation of trees

A simple data structure can be used to represent a tree. We assume in general:

```
typedef TreeNode TreeNode, * TreeNodePtr;
typdef char * NodeInfo;  /* For example */
```

where `TreeNode` is the data type for a node in a tree and `NodeInfo` is the data type for the information associated with each node. The definition of a tree indicates that each node knows all its children. If we had an ADT `TNPBag` that could hold an arbitrary collection of node pointers, the `TreeNode` data structure could be defined as:

```
struct TreeNode {
   NodeInfo info;  /* Data associated with node */
   TNPBag kids;    /* The collection of node's children */
}
```

Other possible data structures that do not require a Bag ADT include:

```
struct TreeNode {
   NodeInfo info;
   int nKids;
   TreeNodePtr kids[];
};
```

```
struct TreeNode {
   NodeInfo info;
   int nKids;
   TreeNodePtr  kids[MAX_KIDS];
};
```

```
struct TreeNode {
   NodeInfo info;
   TreeNodePtr parent;
   int nKids;
   Node * kids[];
};
```

```
struct TreeNode {
   NodeInfo info;
   TreeNodePtr parent;
   int nKids;
   TreeNodePtr  kids[MAX_KIDS];
};
```

The four variations make different choices about whether there is a maximum number of children for any node and whether explicit references to the parent are included in the data structure.

We can initialize data structures such as the one in Figure 7.1 as follows:

```
typedef struct TreeNode TreeNode, * TreeNodePtr;
typedef char * NodeInfo;  /* For example */

#define MAX_KIDS 2
```

```
struct TreeNode {
  NodeInfo info;
  int nKids;
  TreeNodePtr  kids[MAX_KIDS];
};

TreeNode me, mom, dad, momsMom, momsDad, dadsDad, dadsMom;

TreeNode me = {"ME", 2, {&mom, &dad}};
TreeNode mom = {"Mom", 2, {&momsMom, &momsDad}};
   /* ... etc ... */
```

7.4 Traversing Trees

Traversing a tree is a systematic method for visiting each node in a tree. There are two classic ways to do this.

Pre-order: This type of traversal is defined recursively as:

1. Visit the root.
2. Visit each of the children.

Post-order: In this case:

1. Visit each of the children.
2. Visit the root.

The trees in Figure 7.8 give the pre- and post-order traversals of each one.

The implementation of these algorithms is simple:

```
void postOrder(TreeNodePtr t)
{
  int k;
  if(t != NULL) {
    for(k = 0; k < t->nKids; k++)
      postOrder(t->kids[k]);
    printf("%s\n", t->info);
```

```
  }
}

void preOrder(TreeNodePtr t)
{
  int k;

  if(t != NULL) {
    printf("%s\n", t->info);
    for(k = 0; k < t->nKids; k++)
      preOrder(t->kids[k]);
  }
}
```

Another way is:

Breadth-first: Here we visit all nodes at depth 0, then at depth 1, and so on.

We will the depth-first traversal method as well as breadth-first traversal (a variation on pre-order) when we look at *graphs* in Chapter 10.

Another way is *in-order*:

1. Visit the children from left to right as follows:
   (a) Visit the child.
   (b) Visit the root.
   (c) Visit the next child.
   (d) Visit the root.
   (e) Visit the next child and so forth.

This does not correspond to a "pure traversal", where each node is visited exactly once, unless the maximum number of children is no more than two. When there are three or more children, the parent is visited more than once.

7.5 Binary trees

A *binary tree* is a rooted tree defined recursively as follows:

- Consists of nothing (i.e. no nodes at all), **or**
- Consists of a node with two children—a *left child* and a *right child*, each of which are *binary trees*.

Binary trees differ from general trees in three ways:

1. The number of children is defined as being exactly two.
2. The order of the children matters.
3. A binary tree can be completely empty. (A general tree requires at least one node.)

Some terms used with binary trees are:

Null tree: A binary tree with no nodes. (Also called an *empty tree*.)

Full binary tree: A tree where all external nodes have the same depth.

Complete binary tree: A binary tree in which all leaf nodes are at depth *height* or *height*-1, and all leaves at depth *height* are towards the left. A *full tree* is always a *complete tree*, the the converse is not generally true.

Figure 7.9 shows some binary trees.
Figure 7.10 shows some (non-) complete binary trees.

7.5.1 Representing Binary trees

We can represent binary trees using parenthetical notation as we did for ordinary trees. However, because we must distinguish between left and right children, we add the additional convention that a nil child is presented with '`()`'. For example, the binary tree in Figure 7.10 with 6 nodes (A–F) would be represented as:

```
(A (B (D) (E)) (C (F) ()))
```

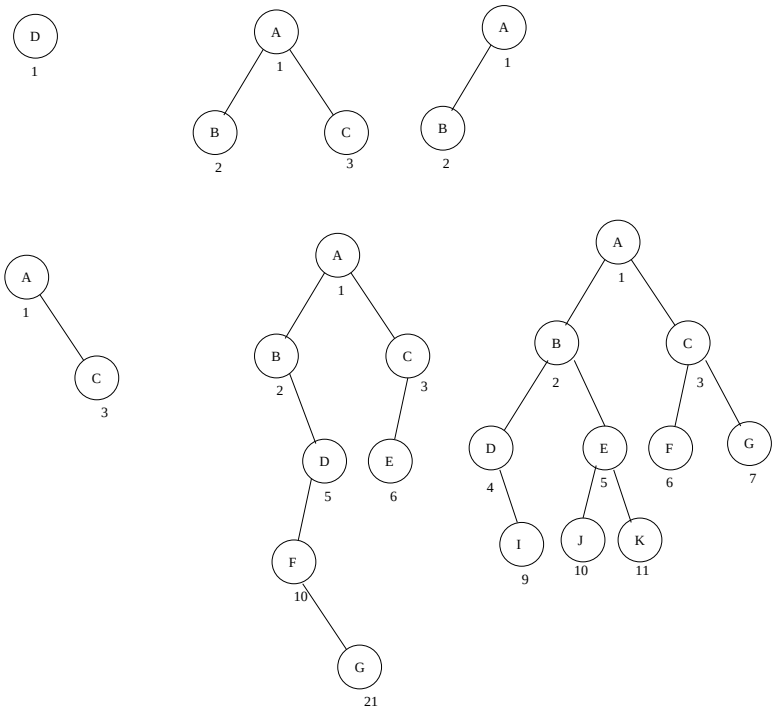


Figure 7.9: Some Binary Trees

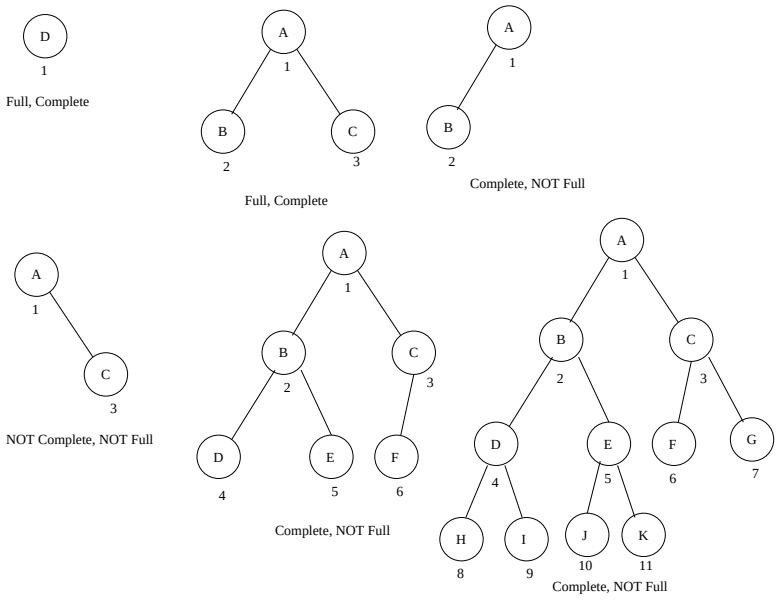


Figure 7.10: Some Complete and Non-complete Binary Trees

Algorithms that use binary trees usually need to use simple operations such as:

GetLeftChild Given a node, get the left child (i.e. the node at the left sub-tree's root.)

GetRightChild Given a node, get the right child.

GetParent Given a node, get the parent node (or NULL if the node is the root of the whole tree.) the left sub-tree's root.)

GetInfo Given a node, get the information associated with the node.

Any representation of a binary tree should make these operations easy to perform; they should be computed in constant time (i.e. independently of the size of the tree.)

Any binary tree can be implemented as a set of linked nodes. Each node in the Binary Tree contains pointers to its left and right child nodes. It is also often convenient to have an explicit pointer to a node's parent. One possibility is:

```
typedef struct BTNode BTNode, * BTNodePtr;
typedef char * NodeInfo;  /* For example */

struct BTNode {
  NodeInfo info;
  BTNodePtr left;
  BTNodePtr right;
  BTNodePtr parent; //Optional
};
```

Array representation of complete binary trees

Complete binary trees can be represented as a simple array of objects containing the information associated with each node. The elements of the array do not need explicit pointers to children or parent.

This is accomplished by numbering each node of a complete binary tree from 1 to n, where n is the number of nodes in the tree. The root node (at depth 0) is numbered 1; its children (at depth 1) are numbered 2 and 3; similarly, the 4 nodes at depth 2 are numbered 4, 5, 6 and 7. In general, the nodes at depth i are numbered from left to right 2^i through $2^{i+1} - 1$.

The node numbered p is stored in element p of the array. It is easy to show that the number of p's left child is simply $2p$ and the number of its right child is $2p + 1$. Similarly, the number of its parent is $p/2$ (using integer division.)

7.5.2 Binary Search Trees

A Binary Search Tree (BST) is a binary tree in which the information associated with each node is called a key and the keys can be ordered. A BST must respect the constraints:

- All nodes to the left of the root must have keys that are smaller than the root's key.
- All nodes to the right of the root must have keys that are bigger than the root's key.
- All subtrees must be BSTs.

Algorithms associated with a binary search tree include:

Find Given a BST, determine if the specified key exists.

Add Add a node with a given key to a BST.

Delete Delete a specified node from a BST.

Max(min) Determine the maximum (minimum) key in a BST.

Successor(predecessor) Determine the next (previous) key from a given node in a BST.

Each of these algorithms is now described.

Find

The *find* algorithm determines if a node exists in a BST. The algorithm is:

Find Algorithm

Search for a node in a BST

Step 1: Start at the root node.

Step 2: If the node is nul, the search is unsuccessful. STOP.

Step 3: If the node information matches the search criterion, the search is successful. STOP.

Step 4: Otherwise, use the same algorithm on the left (right) child tree when the key is smaller (bigger) than the current node.

Add

The *add* algorithm is used to add a node to an existing BST. In a nutshell, the algorithm is:

Add Algorithm

Add a node to a BST

Step 1: Use the Find algorithm to find the node or the nul node where it would go.

Step 2: It it already exists, it cannot be added. STOP.

Step 3: Otherwise, replace the external node with the new node.

Delete

The *delete* algorithm is used to remove a node to an existing BST. In a nutshell, the algorithm is:

Delete Algorithm

Delete a node to a BST

Step 1: If the node as 0 or 1 child, splice it out and STOP.

Step 2: Otherwise (it has 2 children), replace its with its successer and delete the successor node (which is guaranteed to have no more than 1 child) by splicing it out. STOP.

7.5.3 Heaps

A heap is defined as a binary tree that meets the following two criteria:

1. It must be a *complete tree*.
2. The value of the root must be bigger than the value of any descendant node and so on recursively.

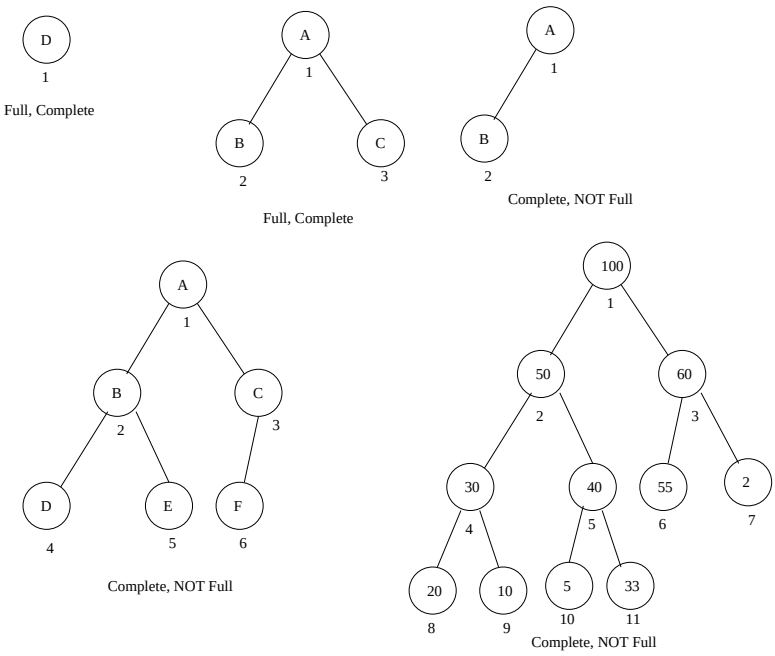


Figure 7.11: Some trees that are heaps

Some heaps are shown in Figure 7.11

Since the biggest value in the tree is in the root node, determining the maximum value is $\Theta(1)$ operation.

FindHeapMax Algorithm

Find the maximum value in a heap

Step 1: Output the value in the root node. STOP.

DeleteHeapMax Algorithm

Find and delete the maximum value in a heap

Step 1: Output the value in the root node.

Step 2: Replace the value in the root node with the value of the last node and delete the last node Ii.e. the rightmost node in the bottom row of the tree.)

Step 3: Interchange the root node value with the value of its biggest child.

Step 4: Continue interchanging until either the bottom of the tree is reached or the node being examined is bigger than both children.

Step 5: STOP.

AddHeap Algorithm

Add a node to an existing heap

Step 1: If the heap is empty, create a 1-node tree with the new node and STOP.

Step 2: Otherwise, add the new node as the last node in the complete tree (i.e. as the next node on the bottom row of the tree if it is not already fully populated; otherwise, as the first–leftmost–node on a new bottom row.)

Step 3: If the new node is bigger than its parent, interchange it with parent.

Step 4: Continue interchanging until either the root of the tree is reached or the node being examined is smaller than its parent.

Step 5: STOP.

7.6 Problems

7.1 Convert the following descriptions or trees (using parenthetical notation) into a standard diagram.

1. `(A (B) (C(D)(E)(F)))`

7.2 Consider the following dependency specification:

```
E: G

A: B C

C: D E F
```

Convert each statement into a tree diagram. Combine the diagrams into a single dependency tree.

7.3 Consider a binary search tree described as:

```
(M (G (D) (K (I) (L))) (R (P) (V)))
```

Draw the tree.
List the nodes:

1. inorder
2. preorder
3. postorder

Using the standard delete algorithm, delete the G node and give the text (parenthetical) representation for the new tree and draw it.

7.4 What are the maximum and minimum possible depth for a tree of n nodes?

7.5 What are the maximum and minimum possible depth for a binary tree of n nodes?

Version 1.1 (2003-03-11) (Chapter version: 2003-04-10)

Chapter 8

Balanced Binary Search Trees

8.1 The problem with ordinary Binary Search Trees(BSTs)

Ordinary BSTs have performance metrics for operations such as "search", "add" and "delete" that are proportional to the height of the tree. If the tree is reasonably balanced, the height is proportional to $\log n$, where n is the number of nodes. Indeed, if the information added to a BST is inserted in random order, the probability that the tree will be reasonably balanced is extremely high. So, what's the problem?

Alas, when information is elicited from mortals, people tend to supply information in an orderly way and—here's the sad part—this tidy way of supplying information is often the very worst way of creating a BST. Specifically, adding information to a BST that is already in sorted order is the worst possible way of creating a tree, but it is likely that people will do precisely that...

Here's a quick example: "Quickly now, tell me the names of 10 numbers in the range $1 \ldots 10$. All of the numbers you name must be distinct." Once the question is understood, the "questionee" is likely to respond, "One, two, three, four, five, six, seven, eight, nine, ten!" Perhaps, they will name the numbers as a count down sequence, "Ten, nine, ... BLAST OFF!". More playful respondents may choose to name even numbers first followed by odd numbers, or maybe they will partition the numbers into primes and non-primes.

However inventive the respondent wishes to be, there is a strong likelihood

that the numbers named will be in at least a semi-sorted order and, hence, our BST will not behave in the way it would if the named numbers were truly randomly selected. In short, it is highly unlikely that someone, when asked this question, will build a 10-sided unbiased die marking each side with a distinct integer in the range $0 \ldots 10$, roll it to get the first number, mark the number so that it should not be used again, roll the die at least 9 additional times and reject results where the number has already been marked as used. Whew... Frankly, if someone were to answer the question in this randomized way, I would be very impressed by their inventiveness, but would harbour some doubts about their sanity.

While there are precisely 3,628,800 (10!) correct ways of answering the question, the solution "One, two, three, four, ..." is likely to occur much more often than would be expected from a random selection of any of the valid answers. (The expection that the sorted answer would be given from a random selection is about .00003%.)

8.1.1 What can we do?

There are ways to change the shape of a BST (while retaining its BST characteristics) that can make the tree more balanced and, hence, obtain the $\Theta(\log n)$ behavorior we desire.

8.1.2 What does "reasonably balanced" mean?

One simple metric for the imbalance of a tree is the difference between the depths of the deepest and shallowes leaf nodes. We call this the "imbalance measure".

One possible criterion for "reasonably balanced" is that the imbalance measure be less than some constant (that must be independent of the number of nodes in the tree). A common requirement is that the imbalance measure must be no greater than 2. (The AVL rebalacing algorithm uses this measure.)

Another possible criterion is that the "imbalance measure" must be less than some (constant) percentage of the tree height. A common requirement is that the ratio between the depths of the shallowest and deepest leaves be less than 2. (The Red-Black algorithm uses this measure.)

8.2 Basic re-balancing methods

Consider the BST in Figure 8.1.

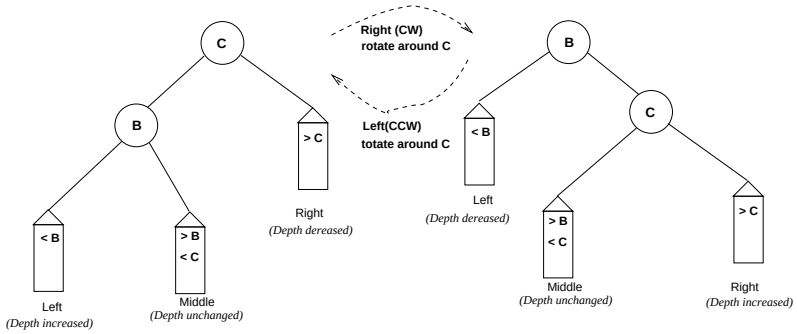


Figure 8.1: A BST with possible imbalances in left or right sub-tree

If the imbalance measure is more than 2 and if the "heavy" subtree is the Left or the Right, then a simple rotation around the root in either the right (clockwise) or left (counter clockwise) direction can reduce the imbalance.

However, when the Middle subtree is the problem, it is a bit more complicated. See Figure 8.2.

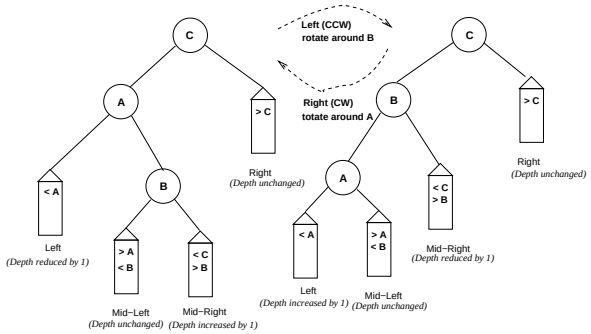


Figure 8.2: A BST with possible imbalances in middle subtree

Version 1.1 (2003-03-11) (Chapter version: 2003-04-10)

8.3 The Red-Black tree algorithm

One algorithm that inserts (and deletes) nodes in a BST and retains enough balance so that all operations have $\Theta(\log n)$ complexity is the Red-Black tree method.

8.3.1 Red-Black Tree (RBT) definition

An RBT is a BST with the additional constraints:

1. Every node is coloured either RED or BLACK.
2. RED nodes can only have BLACK childdren.
3. Leaf nodes are BLACK.
4. The "Black path length" to all leaves is a constant where the "Black path length" to a leaf is defined as the total number of black nodes present in the path from the root to the leaf.

In addition, by convention, we always colour the root BLACK.

8.3.2 RBT insert algorithm

The insert algorithm begins with the standard BST insert method and colours the newly inserted node as RED. (The tree that has a node added to it must be a RBT prior to insertion.)

After this initial step, the new tree will definitely continue to statisfy all of the constraints of an RBT except (possibly) for the requirement that "RED nodes can only have BLACK childdren".

(Note: the other 3 constraints will hold. Clearly since every previoulsy existing node was either RED or BLACK and a new node is added with the colour RED, then every node is coloured. Leaf nodes (i.e. empty binary trees) continue to be BLACK). The "Black path length" cannot change for any leaf since there no BLACK nodes have be added or deleted.)

Consequently, the resulting tree after the simple insertion of a new RED node can only violate the RBT constraints if its parent is also RED.

If the parent is RED, we consider two different possibilities.

RED uncle

First, if the "uncle" is RED (the uncle node is the sibling of the parent), then a simple recolouring will reduce (and perhaps fix) the problem as illustrated in Figure 8.3. (Note that all the RBT constraints continue to apply with the possible exception that recolouring the grandparent RED will make it in violation. Even if this is the case, however, the violation has moved closer to the root and, if the violation were to continue we would eventually reach the root in $\log_2 n$ steps where n is the constant Black path length.)

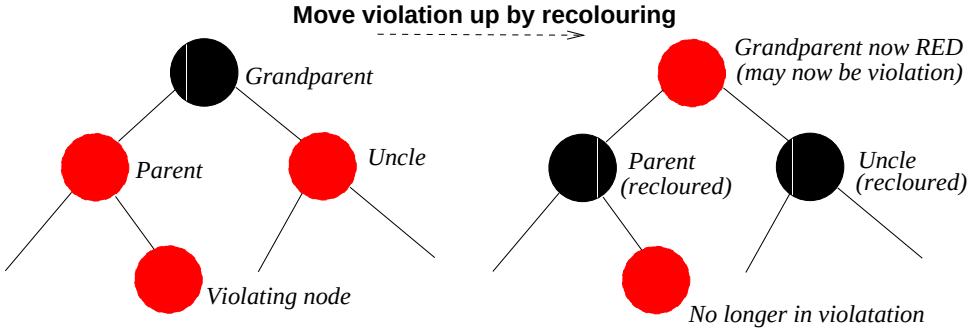


Figure 8.3: Red parent and red uncle after basic insertion

BLACK uncle

In this case, there are two possibilites that are handled differently.

Violating node (with BLACK uncle) is a left child: In this case we need to perform a rotation (right rotation about the parent) and some recolouring as shown in Figure 8.4.

Violating node (with BLACK uncle) is a right child: In this case we need to perform a rotation (left rotation about the violating node)as shown in Figure 8.5.

The net effect is that the there is still a node in violation, but it is now a left child and the previous case applies which is then performed.

Note that the above cases are valid if the parent of the violating node is a left child. The situation is symmetric if the parent is a right child but you need to then interchange the words left and right.

Version 1.1 (2003-03-11) (Chapter version: 2003-04-10)

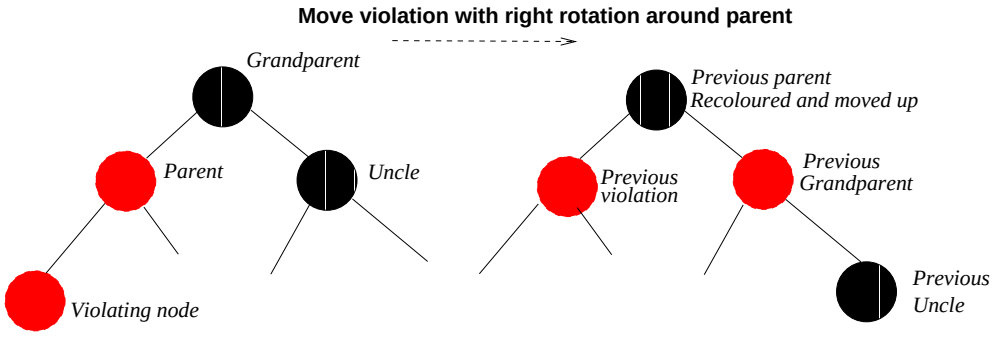


Figure 8.4: Red parent, black uncle, left child

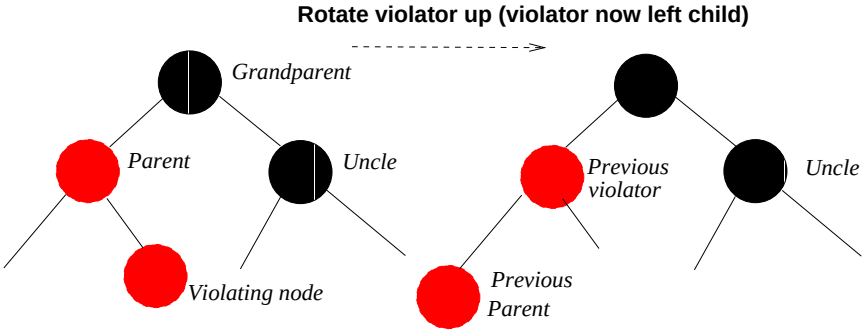


Figure 8.5: Red parent, black uncle, right child

http://yallara.cs.rmit.edu.au/~gregston/CS544/CS544/Terry_Gunning/RBTdemo.html
http://www2.cs.utah.edu/classes/cs3510/applets/RedBlackTree/

Red-black trees are used to implement the Java utility class `TreeMap`.

```
/*
 * @(#)TreeMap.java     1.27 98/05/06
 *
 * Copyright 1997, 1998 by Sun Microsystems, Inc.,
 * 901 San Antonio Road, Palo Alto, California, 94303, U.S.A.
 * All rights reserved.
 *
 * This software is the confidential and proprietary information
 * of Sun Microsystems, Inc. ("Confidential Information").  You
 * shall not disclose such Confidential Information and shall use
 * it only in accordance with the terms of the license agreement
 * you entered into with Sun.
 */

/**
 * Red-Black tree based implementation of the Map interface.  This class
 * guarantees that the Map will be in ascending key order, sorted according
 * to the <i>natural order</i> for the key Class (see Comparable), or
 * by the Comparator provided at TreeMap creation time, depending on which
 * constructor is used.  Note that this ordering must be <em>total</em>
 * in order for the Tree to function properly.  (A total ordering is
 * an ordering for which a.compareTo(b)==0 implies that a.equals(b);
 * see OrderedMap for further details.)
 * <p>
 * This implementation provides guaranteed log(n) time cost for the
 * containsKey, get, put and remove operations.  Algorithms are adaptations
 * of those in Corman, Leiserson, and Rivest's <EM>Introduction to Algorithms</EM>.
 *
 * @author  Josh Bloch and Doug Lea
 */


    /** From CLR **/
    private void fixAfterInsertion(Entry x) {
        x.color = RED;
```

Version 1.1 (2003-03-11) (Chapter version: 2003-04-10)

```
        while (x != null && x != root && x.parent.color == RED) {
            if (parentOf(x) == leftOf(parentOf(parentOf(x)))) {
                Entry y = rightOf(parentOf(parentOf(x)));
                if (colorOf(y) == RED) {
                    setColor(parentOf(x), BLACK);
                    setColor(y, BLACK);
                    setColor(parentOf(parentOf(x)), RED);
                    x = parentOf(parentOf(x));
                } else {
                    if (x == rightOf(parentOf(x))) {
                        x = parentOf(x);
                        rotateLeft(x);
                    }
                    setColor(parentOf(x), BLACK);
                    setColor(parentOf(parentOf(x)), RED);
                    if (parentOf(parentOf(x)) != null)
                        rotateRight(parentOf(parentOf(x)));
                }
            } else {
                Entry y = leftOf(parentOf(parentOf(x)));
                if (colorOf(y) == RED) {
                    setColor(parentOf(x), BLACK);
                    setColor(y, BLACK);
                    setColor(parentOf(parentOf(x)), RED);
                    x = parentOf(parentOf(x));
                } else {
                    if (x == leftOf(parentOf(x))) {
                        x = parentOf(x);
                        rotateRight(x);
                    }
                    setColor(parentOf(x),  BLACK);
                    setColor(parentOf(parentOf(x)), RED);
                    if (parentOf(parentOf(x)) != null)
                        rotateLeft(parentOf(parentOf(x)));
                }
            }
        }
        root.color = BLACK;
    }
```

```
/** From CLR **/
private void rotateLeft(Entry p) {
    Entry r = p.right;
    p.right = r.left;
    if (r.left != null)
        r.left.parent = p;
    r.parent = p.parent;
    if (p.parent == null)
        root = r;
    else if (p.parent.left == p)
        p.parent.left = r;
    else
        p.parent.right = r;
    r.left = p;
    p.parent = r;
}

/** From CLR **/
private void rotateRight(Entry p) {
    Entry l = p.left;
    p.left = l.right;
    if (l.right != null) l.right.parent = p;
    l.parent = p.parent;
    if (p.parent == null)
        root = l;
    else if (p.parent.right == p)
        p.parent.right = l;
    else p.parent.left = l;
    l.right = p;
    p.parent = l;
}

/**
 * Associates the specified value with the specified key in this TreeMap.
 * If the TreeMap previously contained a mapping for this key, the old
 * value is replaced.
 *
 * @param key key with which the specified value is to be associated.
 * @param value value to be associated with the specified key.
 * @return previous value associated with specified key, or null if there
```

Version 1.1 (2003-03-11) (Chapter version: 2003-04-10)

```
     *         was no mapping for key.  A null return can also indicate that
     *         the TreeMap previously associated null with the specified key.
     * @exception ClassCastException key cannot be compared with the keys
     *            currently in the TreeMap.
     * @exception NullPointerException key is null and this TreeMap uses
     *            natural order, or its comparator does not tolerate
     *            null keys.
     */
    public Object put(Object key, Object value) {
        Entry t = root;

        if (t == null) {
            incrementSize();
            root = new Entry(key, value, null);
            return null;
        }

        while (true) {
            int cmp = compare(key, t.key);
            if (cmp == 0) {
                return t.setValue(value);
            } else if (cmp < 0) {
                if (t.left != null) {
                    t = t.left;
                } else {
                    incrementSize();
                    t.left = new Entry(key, value, t);
                    fixAfterInsertion(t.left);
                    return null;
                }
            } else { // cmp > 0
                if (t.right != null) {
                    t = t.right;
                } else {
                    incrementSize();
                    t.right = new Entry(key, value, t);
                    fixAfterInsertion(t.right);
                    return null;
                }
            }
```

```
        }
    }
```

8.4 AVL trees

8.5 Splay trees

8.6 Problems

8.1 Figure 8.6 shows a BST.

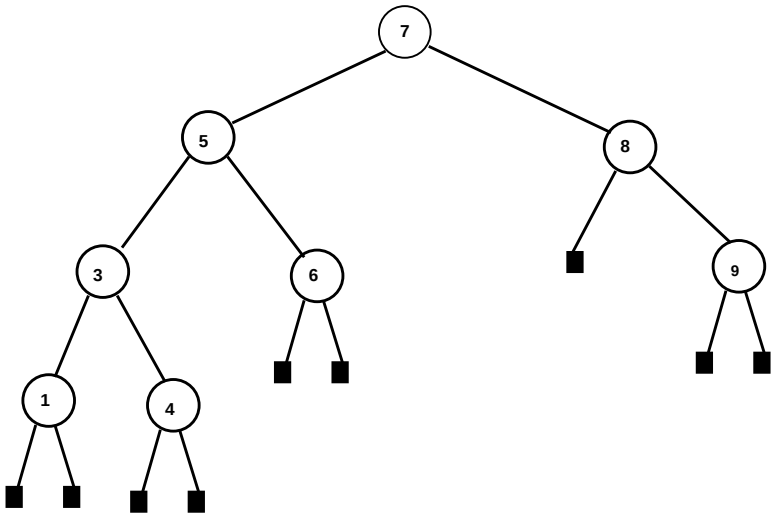


Figure 8.6: A binary search tree

a. What is the path length from the root to the deepest leaf node?

b. What is the path length from the root to the shallowest leaf?

c. Can nodes be colored so that it is a red-black tree? If so, show the coloring.

8.2 A Red-Black BST is initially empty. Nodes with the keys 1, 2, 5, 3, 6, 4 are added in that order. Show the evolution of the tree after each addition.

Version 1.1 (2003-03-11) (Chapter version: 2003-04-10)

Chapter 9

Hash tables

Balanced binary search trees are excellent data structures for maintaining a set of items where the operations *find*, *add*, *delete*, *successor* and *predecessor* are all important in an application.

However, there is a wide number of applications where the *find* operation is the most important and the *add* and (possibly) the *delete* operations are also important but where retrieving the data in sorted order is rarely needed.

In such cases, a *dictionary* type of data structure—usually implemented as a *hash table* is often the most appropriate choice.

As we shall see in this chapter, a properly implemented hash table provides $\Theta(1)$ performance for the find, add and delete operations (compared to the $\Theta(\log n)$ performance of BSTs.)

In this chapter we assume that the items contained in the hash table consist of data structures where one of the fields is unique. For example, if we were dealing with data structures containing information about students in a university, the field uniquely identifying an individual student would be the student number. (In our university, this is a 9-digit number.)

9.1 Mapping data to numbers

Suppose the universe of keys an application deals with is:

```
<key> ::= <consonant> <vowel> <consonant>
<consonant> ::= b|c|d|f|g|h|j|k|l|m|n|p|q|r|s|t|v|w|x|z
<vowel> ::= a|e|i|o|u
```

Version 1.1 (2003-03-11) (Chapter version: 2002-02-27)

In this case, all valid keys would be exactly 3 characters long. Some examples of keys are "dog", "cat", "bin" and "zed". Invalid keys include words like "foo" (last character is a vowel), "foxy" (too many characters) and "do" (too few characters.)

There is also a relatively small number of possible keys: the first letter can be any of 21 consonants, the second any of 5 vowels and the last any consonant. In all, then, there are $21 \times 5 \times 21 = 2205$ possible keys.

All possible keys can then be mapped into a unique integer in the range 0—2204. Furthermore, any integer in the range can be converted back into its corresponding unique key.

Details about this mapping—you can skip this

One way to do this is consider consonants to be mapped to their ordinal (i.e. a number in the range 0—20); similarly, each of the 5 vowels are mapped to 0—4. We can then consider the key to be represented as a mixed-base number.

For example, consider the key "dog" (consonant 2, vowel 3, consonant 4):

$$\text{dog} \implies 2_{21}3_{5}4_{21} = 2 \times 21 \times 5 + 3 \times 21 + 4 = 277$$

Similarly, we can convert the mapped integer back to the key. For example, if the integer is 121, we have:

$$\text{first consonant ordinal } = 121/105 = 1(\text{ remainder} = 16)$$

$$\text{middle vowel ordinal } = 16/21 = 0(\text{ remainder} = 16)$$

$$\text{last consonant ordinal } = 16$$

Since consonant 1 is 'c', vowel 0 is 'a' and consonant 16 is 't', the integer 121 is the mapped value of the key "cat".

The main characteristic of this example is that the total universe of possible keys is quite small. Because of this, it is possible to map each possible key into an integer in the range 0—$n-1$ where n is the size of the key universe.

Furthermore, a reverse mapping is also easily done and both mapping and its inverse can be done in constant time ($\Theta(1)$.)

Hence we can allocate an array of n bits where each bit indicates whether the mapped key exists or not. Initially, each bit is set to `false` and as keys are added, the corresponding bit is set to `true`. Determining if a key exists is trivial: calculate the map function (in $\Theta(1)$ time) and look at the corresponding bit.

9.2 Hash tables

The assumption made in case of direct-mapped tables are rarely met. In real applications, the key field of a data item often has a *huge* number of possibilities. For example, the 9-digit student numbers used at our university allow for up to 1 billion students—there are much fewer.

Hash tables have some similarities to direct mapped tables in that each key must be unique and keys are converted to an integer that is used as an index into a table where the item can be found. However, unlike direct mapping, the mapped integer (called the *hash code* does *not* have a one-to-one mapping to keys. (Hence, there is no way to convert a hash code back into a key.) On the contrary, several different keys can have the same hash code.

Since knowledge of the hash value of a key (i.e. the slot number for that key) is insufficient to identify the key, we cannot simply use boolean values in slots to indicate that it is used or not. When the slot is used, we must also keep track of the specific key that occupies the slot. This is the first difference between a direct-mapped table and a hash table.

The second difference—and the one that makes the hash table algorithms a bit more complex than their direct-mapped table counterparts—is how *collisions* (two or more keys that have the same hash code) are handles.

To summarize:

- A hash table is an array of slots (numbered 0 to size-1).
- The size is usually a prime number.
- The data to be stored converted to a slot number with a "hash function". If that slot is empty, the data is placed there. Otherwise, a *collision* occurs.

Version 1.1 (2003-03-11) (Chapter version: 2002-02-27)

- A commonly used hash function is $h(key) = key \bmod size$

An important parameter that influences performance of hash tables is the *load factor*(α) defined as:

$$\alpha = \frac{\text{number of slots}}{\text{number of items}}$$

9.2.1 Collision resolution by chaining

In this method:

- Each slot is the head of a (possibly empty) list of data added to the hash table whose hash function identifies the slot.

9.2.2 Collision resolution by probing

In this method:

- The next slots are tried until an empty slot is found.
- To search for an item, the natural slot is tried. If it is not empty and the data is not what is being looked for, consecutive slots are examined until either the data is found or an empty slot occurs.

9.2.3 Collision resolution by double hashing

- If the slot is occupied, additional slots are attempted by going through the table in increments defined by the secondary hash function.
- It is common to use $1 + key \bmod size - 1$ as the incrementer.
- For example, to add key 20 to a hash table of size 11, calculate the natural slot ($28 \bmod 11 = 6$). If a collision occurs, calculate the double hash increment ($1 + 28 \bmod 10 = 9$). The try $6 + 9 \bmod 11 = 4, 6 + 2 * 9 \bmod 11 = 2$, etc. until an empty slot is found.

9.3 Problems

9.1 A Hash Table implemented in an array of size 13 is initially empty and the following integer keys are added in the order given: 5, 19, 17, 18, 4, 6. Assume the primary hash function is:

$$h(key) = key \pmod{13}$$

1. Draw a representation of the Hash Table when chaining is used for collision resolution.

2. Draw a representation of the Hash Table when linear probing is used for collision resolution.

3. Draw a representation of the Hash Table when double hashing is used for collision resolution. Assume that the secondary hash function is

$$h_2(key) = 1 + key \pmod{12}$$

Version 1.1 (2003-03-11) (Chapter version: 2003-03-30)

Chapter 10

Graphs

Graphs and Trees—General concepts

10.1 Some definitions

10.1.1 Graph terminology

A graph consists of a set of *nodes* connected by *arcs*. Some examples of graphs are shown in Figure 10.1.

Node: Nodes are the objects that are connected in a graph. Nodes are also referred to as *vertices*.

Arc: A connection between one node and another node (which may be the same as the first node; in this case, the arc is called a *self loop*). Arcs are also referred to as *edges* or *connections*.

Directed graph: If the arcs are directed (indicated with an arrow at one end of the arc), the graph is called a directed graph (or *digraph*). Otherwise, the graph is *undirected*.

Degree: The number of arcs connected to a node in an undirected3 graph is called the node's *degree*

In(out)-degree: For digraphs, the number of arcs directed away from a node is called the node's *out-degree*; the number of arcs entering it are its *in-degree*.

Path: A sequence of arcs starting at one node and terminating on another (possibly the same) node.

Path length: The number of arcs traversed in a path.

Reachable: One node is *reachable* from another if there is a path between the two nodes.

Simple path: A path in which no node is encountered more than once.

Cycle: For undirected graphs, a cycle is a path of length 3 or more with the same starting and ending nodes and where no other node in the path is visited more than once. A *simple cycle* is one with no intermediate node visited more than once. For directed graphs, a cycle is any path with the same starting and ending nodes. A digraph with no self-loops is *simple*.

Connected: If there is a path between every node and any other node, the graph is *connected*.

10.1.2 Free trees

Free tree: A *free tree* is a connected, acyclic, undirected graph.

Free trees have the c following properties:

1. There is a unique path between any two nodes in the tree.
2. If any arc is removed, the graph is no longer connected and, hence, no longer a tree.
3. The number of arcs is one less than the number of nodes.
4. If any arc is added, the graph becomes cyclic and, hence, is no longer a tree.

10.1.3 Rooted trees

A *rooted tree* is a tree in which one node is called the *root* and is the starting point for reaching all other nodes.

Root: The node at the "top" (or "bottom") of the tree.

Parent and Child: The nodes directly connected to a node and below it are called the node's children and the node itself is called the parent. The parent of a parent is called the *grandparent*.

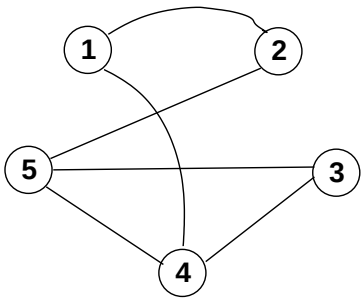
Siblings: Children of the same parent are called *siblings*.

Internal node: Nodes with children.

External node: A node with no children (at the "bottom" of the tree).

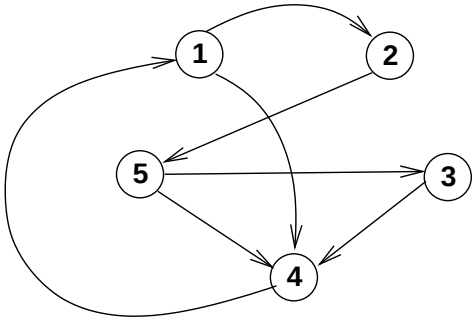
Depth: The depth of the root is 0 (zero). Its children are at depth 1; its grandchildren are at depth 2, etc.

Height: The maximum depth of a tree.



(a) An undirected graph with:

- 5 nodes
- 6 arcs
- connected
- cyclic



(a) A directed graph with:

- 5 nodes
- 7 arcs
- connected
- cyclic

Figure 10.1: Some graphs

10.2 Graph representations

There are two common ways to represent a graph:

Adjacency-list: A linked-list of adjacent nodes is maintained for each node of the graph.

Adjacency-matrix: A $V \times V$ boolean matrix (where V is the number of vertices in the graph) is used. Each element in the matrix indicates the presence of an edge between the nodes corresponding to the row and column of the element.

10.3 Traversal algorithms

```
1: Color all nodes white.
2: Choose a start node, color it gray and add it to GrayBag.
3: repeat
4:    Set next ← node removed from GrayBag
5:    Set next color to Black.
6:    for all White nodes adjacent to next do
7:       Color each one Gray and add to GrayBag.
8:    end for
9: until GrayBag is empty.
```

In this algorithm, we use the colors White, Gray and Black do distinguish the nodes:

White: Nodes that have not yet been examined at all.

Gray: Nodes that we have begun to examine.

Black: Nodes that have been completely examined.

We can consider the Gray nodes to represent the boundary between the nodes we have examined and ones that have not yet been looked at.

In this generic algorithm, we do not state the ordering characteristics of the "GrayBag". Recall that we use the abstract term "Bag" for a data structure that can hold an arbitrary collection of things; we can only add or remove one item at a time from the collection.

Common types of Bags include Queues, Stacks and Priority Queues. The precise behavior of the algorithm depends on what specific kind of bag we use. The most commonly used collections for this traversal algorithm are Stacks and Queues. Note that in both of these cases adding or removing an item from the collection is a constant time ($\Theta(1)$) operation.

What is the complexity of this algorithm?

Note that in the outer repeat loop, one node is colored Black each time through the loop. Once a node is Black, it can no longer be added to the Bag (since only White nodes are added and immediately colored Gray.) Once no more nodes can be colored Black, the algorithm terminates.

Consequently, the outer loops executes at most V times (where V is the number of nodes.)

The inner loop (finding adjacent White nodes) can execute at most E times in all (where E is the number of edges in the graph.)

All of the other steps have $\Theta(1)$ complexity (assuming the Bag is either a Queue or a Stack). Consequently, the worst case complexity of the algorithm is $\Theta(V + E)$.

We now consider the behavior of the algorithm when a Stack or a Queue is used as the container for Gray nodes.

10.4 Breadth first search

The BFS algorithm is as follows:

```
1: Color all nodes white.
2: Choose a start node, color it gray and add it to Queue.
3: Set start.parent ← NIL.
4: Set start.depth ← 0.
5: repeat
6:    Set next ← node removed from Queue
7:    Set next color to Black.
8:    for all White nodes adjacent to next do
9:       Color each one Gray, add to Queue.
10:      Set each one's parent ← next.
11:      Set each one's depth ← next.depth + 1.
12:   end for
13: until Queue is empty.
```

In this case, we also generate a tree structure (with the start node as the root) and record the depth of each node in the tree.

Version 1.1 (2003-03-11) (Chapter version: 2003-03-30)

10.5 Depth first search (DFS)

The DFS algorithm is as follows:

1: Color all nodes white.
2: Choose a source node, color it gray and add it to Stack.
3: Set prev $\leftarrow$ NIL.
4: Set source.start $\leftarrow$ 0.
5: **repeat**
6: Set *next* $\leftarrow$ node at top of Stack
7: Color each one Gray, add to Stack.
8: Set each one's depth $\leftarrow$ $next.depth + 1$.
9: **until** Stack is empty.

10.6 Topological Sort

1: Do a DFS
2: Front of linked list when finished.

10.7 Weighted Graphs

Weighted graphs have a weight (or cost) associated with each edge.

10.8 Minimum Spanning Tree

A Minimum Spanning Tree (MST) is the set of edges that connect all nodes with the lowest cost.

10.8.1 Prim's algorithm

all keys infity except source 0
All all nodes to PQ (min)
repeat
next = minPQ
for all adj **do**
if key > weight, set key to weight and set parent
end for

until empty

10.9 Shortest distance

In this section we look at algorithms to determine the minimum cost to get from one node to any other rachable node.

10.9.1 Relaxation

10.9.2 Dijkstra's algorithm

This algorithm only applies to graphs where all the weights are non-negative.

The algorithm is:

all keys infinity except source which is zero.
All all nodes to PQ (min)
repeat
  next = minPQ
  for all adj to next **do**
    if adj.key > weight+next.key, set adj.key to weight+next.key and set parent
  end for
until PQ is empty

10.9.3 DAG shortest path

10.10 Problems

10.1 An undirected graph with 6 nodes labeled "A", "B", "C", "D", "E" and "F" is described by the following adjacency list:

The entries in the adjacency list column give the nodes connected to the one named in the first column. For example, the entry "B" in the first row means there is an arc between "A" and "B".

1. Draw the graph.

2. Using the standard breadth first search algorithm starting at A, state the order of nodes visited.

Node	Adjacency list
A	B F
B	A C E
C	B D E
D	C E
E	B C F D
F	A E

3. Repeat the above for depth first search.

Chapter 11

Computational theory

This chapter does not yet exist—I am working on it...

OUTLINE April 8, 2004

OUTLINE April 8, 2004

Chapter 12

Strategies

This chapter does not yet exist—I am working on it...

OUTLINE April 8, 2004

OUTLINE April 8, 2004

Chapter 13

Algorithms in Hardware

This chapter does not yet exist—I am working on it...

OUTLINE April 8, 2004

OUTLINE April 8, 2004

Part III

Projects

OUTLINE April 8, 2004

Chapter 14

A Simple Digital Circuit Simulator Engine

This is a quick overview of how *event-driven simulators* work. We first *analyze* the problem, then we *design* the public interface to the required data types, and, finally, we give a partial *implementation* of the design. Students in ELE 428 are expected to study this document on their own to understand the requirements of Lab 5. Understanding and doing the exercises should be sufficient background for tackling the lab.

14.1 How to simulate a digital circuit (*Analysis*)

The event-driven model of digital circuit simulation presented here assumes that the circuit is composed of *blocks* that are interconnected by *wires*. Each wire has a valid value at all times. When the value of a wire changes, all of the blocks that use the wire as an input re-evaluate themselves to determine if the new input conditions require a change to the value of any of their output wires. When an output wire will change, the "block evaluation" creates a *change event* indicating what wire will change its value and the time that the change should occur. This "change event" is then added to a priority queue.

The simulator implements the following algorithm:

Step 1: If there is nothing in the Event Queue, STOP.

Step 2: Get the next event.

Step 3: Determine all the blocks that use the wire in this event as an input and re-evaluate them.

Step 4: Go back to *Step 1*.

This terse, but accurate, description of an event-driven simulator may not be completely satisfying. The following examples flesh out the details of what the simulator actually does.

14.1.1 An ideal 2-input AND gate

Consider the simple AND gate in Figure 14.1.

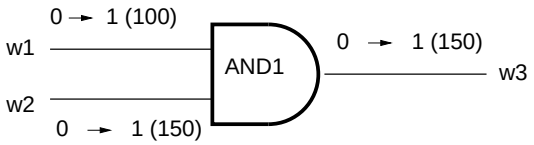


Figure 14.1: A simple AND circuit

We begin the simulation of this simple circuit—consisting of 3 wires (w_1, w_2 and w_3) and a single block(AND_1)—by first assuming that all wires initially are at zero and ensuring that this is a consistent configuration.

We also assume that an event queue has been set up up containing 2 events:

1. Wire w_1 becomes 1 at time 100.

2. Wire w_2 becomes 1 at time 150.

We remove the item with the least time from the event queue and see that w_1 has changed. We then determine that the block AND_1 uses this wire as an input; so we re-evaluate the output that the block will now produce. Since the output is still zero, nothing further occurs in response to this event.

We then remove the next event which tells us that w_2 has become a 1 at time 150. Once again, block AND_1 evaluates; this time, however, it determines that its output wire, w_3, should change to a 1; hence it adds a

new event to the queue indicating that w_3 becomes 1 at time 150. (Note: we are assuming an AND gate with no delay.)

We now remove the next event from the queue (the same one we just added). In this case, w_3 is not connected to the input of any block, so there is nothing to do with the event.

Finally, since there are no longer any events in the queue, the simulation terminates.

14.1.2 An ideal AND followed by a delay

Now, consider an ideal AND gate followed by a delay as shown in Figure 14.2.

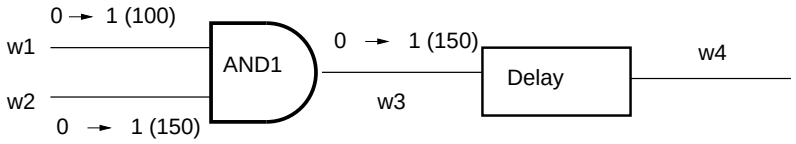


Figure 14.2: An ideal AND followed by a delay

We also assume that an event queue has been set up up containing 3 events:

1. Wire w_1 becomes 1 at time 100.
2. Wire w_2 becomes 1 at time 150.
3. Wire w_2 becomes 0 at time 200.

When the simulation algorithm is executed, we obtain the following:

1. The next event (w_1 becomes 1 at time 100) is removed from the Event Queue.
2. The AND gate is re-evaluated, but no further events are generated.
3. The next event (w_2 becomes 1 at time 150) is removed from the Event Queue.
4. In this case, the output of the AND is scheduled to change; a new event (Set $w_3 \leftarrow 1$ at time 150) is generated.

   The event queue now contains:

   (a) Wire w_3 becomes 1 at time 150.

   (b) Wire w_2 becomes 0 at time 200.

5. The next event (w_3 becomes 1) is now removed from the Event Queue.

6. Since w_3 is an input to the 10-unit delay, the "delay" is re-evaluated, and adds a new event to the Event Queue indicating that w_4 becomes 1 at time 160. After this Event Queue removal and addition, it looks like:

   (a) Wire w_4 becomes 1 at time 160.

   (b) Wire w_2 becomes 1 at time 200.

7. The event "w_4 becomes 1" is next removed from the Event Queue. Since this wire has no dependent blocks, nothing further occurs in the handling of this event.

8. The event "w_2 becomes 0 at time 200" is next removed. Since w_2 does have a dependent block (i.e. the AND gate), it is re-evaluated and adds an event (w_3 becomes 0 at time 200) to the Event Queue which now looks like:

   (a) Wire w_3 becomes 0 at time 200 .

   (b) This results in the creation of the event "w_4 becomes 0 at time 210.

   (c) When this event is removed, no further events are generated and the simulation terminates.

14.1.3 A clock generator circuit using an INVERTER

Consider now a circuit with feedback—i.e. when an output wire is connected back to an input. The simplest example is an INVERTER with its output connected to its input as shown in Figure 14.3 below:

In this case, we will assume that the INVERTER has an embedded propagation delay (i.e. we assume that the INVERTER described here is really an ideal inverter followed by an ideal delay element).

Assume that the clock signal is initially zero. Since the output of the inverter should be a "1", an event is scheduled to occur setting the signal to

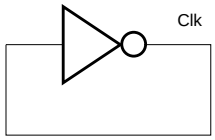


Figure 14.3: A clock generator

"1" at time 10 (assuming the propagation delay of the inverter is 10). When this event is removed from the queue, the gate is re-evaluated, and a new event setting the clock to "0" at time 20 is added to the event queue. This continues forever.

14.1.4 A clocked D-latch

Consider Figure 14.4 below for a clocked D-latch constructed with NAND gates.

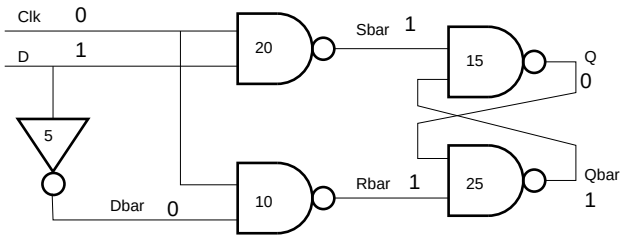


Figure 14.4: A Clocked D-Latch

Each node is identified by a name (such as Sbar) and an initial stable value (1 or 0). You should convince yourself that the node values are indeed consistent and that the circuit will remain in the indicated state forever if neither of the inputs (Clk and D) change.

The numbers in the NAND gates in Figure 14.4 are not just labels; they are meant to represent the propagation delay in nanoseconds of the gate. Using this fact, we can describe the evolution of the circuit in greater detail if Clk becomes 1 at time 0 as follows:

1. The Sbar output of the NAND-20 gate goes to 0 at 20ns.

2. The output of NAND-15 (Q) becomes 1 at 35 ns.

3. The change in Q—an input to NAND-25—from 0 to 1 causes the Qbar output to change from 1 to 0 at 60 ns.

4. The Qbar change is fed back to NAND-15, but its output is not modified by this input change; the circuit is now stable.

The timing is shown in Figure 14.5 below.

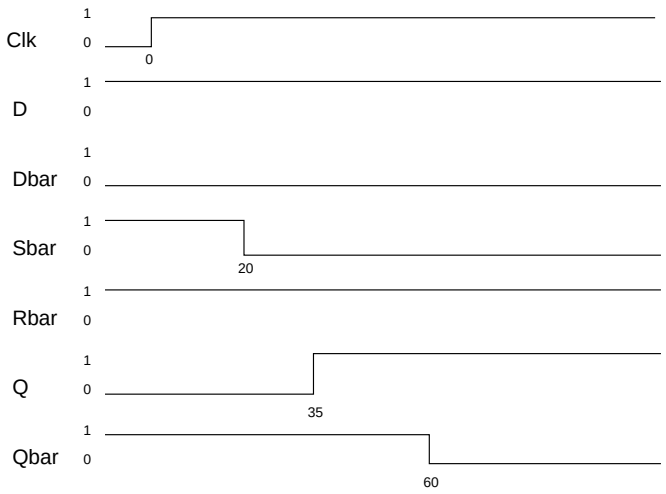


Figure 14.5: A Clocked D-Latch Timing Diagram

14.1.5 Generating Fibonacci Numbers in hardware

Consider Figure 14.6 below which generates the Fibonacci series in hardware.

14.2 The data types needed (*Design*)

14.2.1 The overall algorithm

We first express the overall algorithm for circuit simulation as a step-by-step algorithm in English. We list the major steps in the outer ("top-level") list; inner lists provide more details about the major steps. The algorithm is also

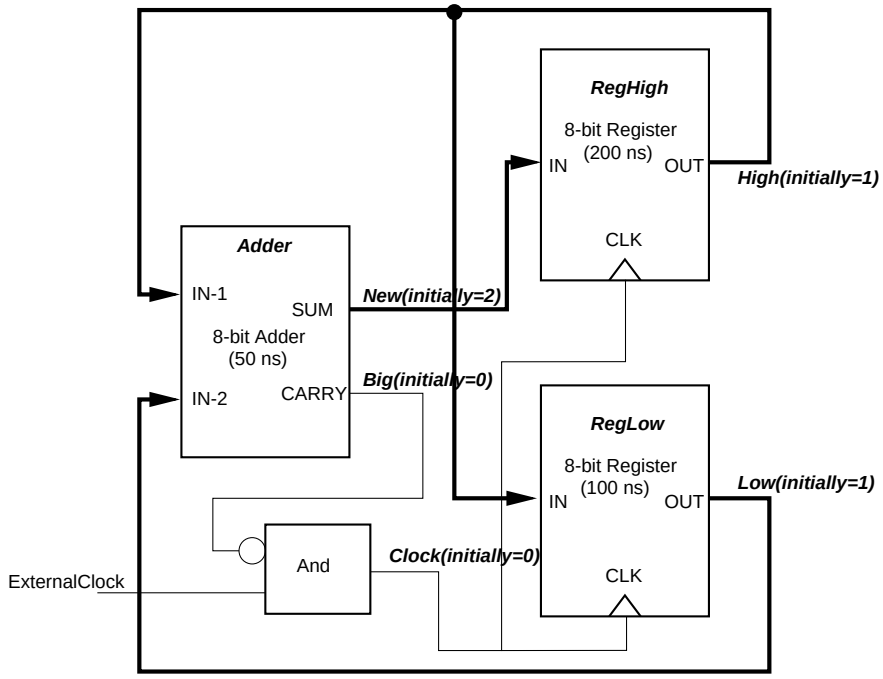


Figure 14.6: Fibonacci number generator

annotated to help us properly design the various data structures. When the algorithm uses a particular data type that needs to be designed, the data type is indicated in *this font*.

The algorithm can be implemented in many different programming languages (such as C, C++, Java, scheme, etc.)

1. Create an empty *event queue*.
2. Create the circuit:
   (a) Create the *wires*: Each wire will be given an initial valid *value* and may also be given a name.
   (b) Create the *blocks*:
      i. Each *block* will know its input and output *wires*.
      ii. When a *block* is created, it will add itself to the dependent blocks of all of its input wires.
3. Add *events* to the *event queue*.

   Note that a *block* may add events to the *event queue* at the time the block is created (if, for example, the current outputs are inconsistent with the inputs).
4. Simulate the circuit.
   (a) If the *event queue* is empty, STOP.
   (b) Remove next *event* from the queue.
   (c) Determine the *wire* set by the *event*.
   (d) If the *wire* does not change its *value*, ignore the event and go back to the first step.
   (e) For all blocks that use the wire as in input:
      Re-evaluate the *block* (possibly adding new events).
   (f) Go back to the first step.

   The remainder of the design will be expressed using C syntax, instead of the more general English description used here.

14.2.2 Value

Value is a very simple data type. Normally, it will just be an integer. The only function related to *value* is:

`int isValidValue(Value_t):` Returns "true" or "false" depending on whether the passed value is legal.

14.2.3 Event

The *event* data type has the following API ("Application Programmer Interface"):

`Event_t newEvent(Wire_t w, Value_t v, int time):` Creates and returns a new *event* for the specified *wire*, value and time.

`Wire_t getEventWire(Event_t e):` Returns the *wire* associated with the *event*.

`long getEventTime(Event_t e):` Returns the time of the *event*.

`Value_t getEventValue(Event_t e):` Returns the value the *wire* takes on at the time of the *event*.

14.2.4 Event Queue

The *event queue* is of central importance to the simulation algorithm. It is very much like a priority queue, except that removing an item deletes the one with the *smallest* key, rather than the *largest* key as a priority queue does. In addition, we will use the *event queue* to keep track of the current simulation time (i.e. the time of the most recently removed event.)

The public API is:

`EventQ_t newEventQ(void):` Creates and returns a new, empty event queue.

`Event_t removeNextEvent(EventQ_t e):` Removes and returns the next *event* in the queue.

`void addEventToQ(EventQ_t eQ, Event_t ev):` Adds the specified *event* to the queue.

`unsigned int eventQSize(EventQ_t eQ);` Returns the number of events currently in the queue.

Note that in our implementation of the *event queue*, we use an existing Priority Queue abstract data type. The implementation section that follows (Section E.6.8) explores this in greater detail.

Priority Queue

The standard priority queue abstract data type is used with the following API:

`PriorityQ_t newPriorityQ(void);` Creates and returns a new, empty priority queue. Items in the queue will be *events*.

`Event_t removeMaxFromPQ(PriorityQ_t pq);` Removes and returns the *event* with the largest time value.

`void addToPQ(PriorityQ_t pq, Event_t e);` Adds the specified *event* to the priority queue.

14.2.5 Wire

Wires are one of the more complex data types. The API is:

`Wire_t newWire(char * name, Value_t initVal);` Creates and returns a new *wire* with the specified name and initial value. The `name` may not begin with "0x". If the name is NULL or the empty string (`""`), a unique name beginning with "0x" will be automatically assigned. Note, there is no guarantee that a user-supplied name will be unique.

`void addWireDependent(Wire_t w, Block_t b);` Adds the named block to the list of blocks that use the wire as an input.

`Value_t getWireValue(Wire_t w);` Returns the current value of the wire.

`void setWireValue(Wire_t w, Value_t v);` Set the current value of the wire to the value specified.

`void resetWireDependents(Wire_t w);` This is used in conjunction with `getNextWireDependent` so that all the dependents will be returned one by one.

`Block_t getNextWireDependent(Wire_t w)`: Returns the next block that uses the wire as an input. When there are no more dependents, NULL is returned. Note that the order in which is the dependents are returned is implementation dependent. Do *not* assume is the same as, or even related to, the order in which the dependents were added.

14.2.6 Block

The *block* abstract data type needs to act as the interface to any specific kinds of blocks. The *block* needs to know its input and output *wires* and how to evaluate itself.

The API is:

`Block_t newBlock(char * name)`: Creates and returns a new block with the specified name. The rules for specifying a name are the same as those for wires.

`void evaluateBlock(Block_t b)`: Causes the block to re-examine its inputs and generate any events on its outputs that are appropriate.

`void addInputToBlock(Block_t b, Wire_t w)`: Adds the specified wire as an input to the block.

`void addOutputToBlock(Block_t b, Wire_t w)`: Adds the specified wire as an output to the block.

14.2.7 Nand

The *nand* data type is a specific implementation of the *block* interface. It implements a "nand" gate that can have any non-zero number of inputs and precisely one output. The output evaluates to 1 if any of the inputs are 0.

The API is:

`Block_t newNand(char * name, unsigned int delay, Wire_t w1, ...)`: Creates a new nand gate with the specified name and propagation delay. At least two wires must be given as additional arguments. The last wire will be the output; all the other wires will be inputs. The last argument (i.e. the argument following the output wire argument) must be `NULL`.

14.3 Exercises

1. Manually determine all of the events that will occur in the Fibonacci circuit (Fig. 14.6). (Assume that the external clock is a square wave with a period of 1000 ns.)

2. Consider the clocked D-Latch in Figure 14.4. Suppose the Clk becomes 1 at time 0 and D becomes 0 at time 24. Determine the sequence of events.

3. Write the `main` function to simulate the clocked D-Latch of Figure 14.4.

4. Sketch out (in pseudo-C) how you would implement `Block_t`s to implement the 8-bit adder and registers used in the Fibonacci generator circuit.

Chapter 15

Combinational Logic

OUTLINE April 8, 2004

Appendix A

Coding Standards

The source code for programming projects should always be organized and written with the future tasks of testing, debugging and maintenance (possibly by others) in mind. These tasks will be easier if the project is well organized and the source code is written in a clear and consistent fashion. In addition, the future possibility of porting the program to different environments (portability) should be addressed at the outset.

This appendix describes some basic rules for project organization that readers of this book may wish to adopt (especially for programming projects suggested in the problems and case studies). I also describe the coding and organizational conventions I used in writing the source code for the book. Finally, I discuss some general aspects of the portability problem.

A.1 Recommended Organizational and Coding Standards

Organization: Each project has a separate directory. For example,

> Each project directory must have `README` and `Makefile` files. The README file should give a general overview of the project and the files that implement it.

Documentation-I (public): The "public" documentation of your source code should inform a reader of who wrote the code and describe *what* it does and how to *use* the interfaces described. The public comments should provide sufficient information for a reader to use the functions

without having to read the actual C code that implements the functions. It is strongly recommended that you write these public comments *before* you write your code. (Writing a function header comment focuses your mind on what you really want the function to do.)

All source code files (*.c and *.h) should conform to the following general commenting standards:

- Identify your work with, for example, a Copyright notice including your name and userid. For example:

```
/* Copyright (C) 1999 Jane Smith (jsmith@ryerson.ca) */
```

- The next comment—the *header*—briefly describes the purpose of the file. For example, a file called `towers.c` might have the following header comment:

```
/**
* The functions in this file solve the classic
* towers of Hanoi problem.
*/
```

- Each function (in a .c file) is preceded by a *function header* comment that briefly describes what the function does as well as indicating the nature of any passed parameters or return value. For example:

```
/**
 * ``main'' manages the command line interface to solving
 * the towers of Hanoi problem.  The command line arg
 * (which must be string representations of numbers)
 * indicate the number of disks to be moved and the source
 * and destination tower numbers. (The towers are
 * identified with the numbers 1, 2 and 3.)
 *
 * @param  argc  the number of command line arguments
 * @param  argv  a pointer to an array of strings where:
 *      There must be exactly 3 arguments where:
 *             -- the fist arg is the number of disks to move
 *             -- the second is the ID-num of the source
 *             -- the third is the ID-num of the destination
```

```
 * @return   always returns an exit code of 0.
 */
int main(int argc, char * argv[])
 {}
```

Notes

The conventions used for the public comments, specifically the `/**` (with the extra `*`) and the tags `@param` and `@return`, correspond to Java commenting standards. In particular, a Java tool called *javadoc* can parse these specially formatted comments and the following declaration to produce nicely formatted HTML documentation automatically. While there is no version of *javadoc* for C code at this time, it does no harm to use the clear conventions of Java in your C code.

Documentation-II (private): While the public documentation should be written so that it does not require the reader to understand or even look at the implementation, private documentation is meant to help the reader understand the actual C code implementing a function. The comments should be written under the assumption that the reader is a competent C programmer. For example:

```
i++;  /* Increment i by one */
```

is a useless comment since it is entirely obvious to a C programmer.

Often, no private comments are required at all in well written programs. The use of descriptive variable and function names is also a great help. Indeed, Rob Pike states:

> Basically, avoid comments. If your code needs a comment to be understood, it would be better to rewrite it so it's easier to understand.
>
> Rob Pike[Pik]

Using descriptive names often eliminates the need for comments, Consider:

```
foo = foo->bar;  /* move "foo" to next item */
```

The comment would be unnecessary with the more intelligent variable and field names:

```
item = item->next;
```

Avoid "magic numbers": Numbers should rarely be placed directly in the source code. (Common exceptions are the numbers 0 (zero), 1 or −1.) Instead use an `enum` data type or the `#define` preprocessor directive. (It is usually preferable to use an `enum` for a small number of integers instead of a `#define`.)

For example, do *not* write code like:

```
double x = 3.14159265358979323846*2.6*2.6;
```

or

```
for(i = 32; i < 212; i += 2)
```

or

```
if ((j = foo()) == 2)
```

instead, use:

```
#include <math.h>  /* This defines the value of PI */
#define RADIUS 2.6
double x = M_PI*RADIUS*RADIUS;
```

or

```
/* Note following temperatures assume Farenheit scale */
#define FREEZING 32
#define BOILING 212
#define TEMP_INCREMENT 2
for(i = FREEZING; i < BOILING; i += TEMP_INCREMENT)
```

or

```
typedef enum {FooGood = 0,
              FooWarn = 1,
              FooBad = 2} FooReturn_t;
if ((j = foo()) == FooBad)
```

Compile with all warnings turned on: You should compile C source code will all warnings turned on. Your C code should produce **no** warnings.

Portability You may only use POSIX/ANSI compatible library functions.

No gotos You can use all of the ANSI C language *except* for the `goto` statement. (The single exception to this rule involves the study of "tail-recursion elimination".)

Header files: Header files should only contain declarations (such as `typedef`s or function prototypes) and preprocessor directives (such as constants and macros). Executable C code (such as a function body) should *never* be placed in a header file.

.h protection All .h files must be protected so that they are never included more than once and that the order of their inclusion is less critical. For example, the header file `foo.h` should be structured as:

```
#ifndef FOO_H
#define FOO_H

/* Body of foo.h with other #includes...  */

#endif /* FOO_H */
```

Line length and avoiding tabs: No source code line should be longer than 80 characters. Use spaces, not tabs, for indentation.

A.2 Other C programming conventions

A.2.1 The *eprintf* library

You may note functions such as `eprintf` or `emalloc` sprinkled through the C code. These functions come from Kernighan and Pike's[KP99, p. 109–111] utility library which we have called `eprintf.o`.

Their use is summarized in Table A.1.

K&P name	"Almost" like	Comments
`eprintf(...)`	`fprintf(stderr, ...)`	Exits
`weprintf(...)`	`fprintf(stderr, ...)`	Prints warning
`emalloc(...)`	`malloc( ...)`	Exits on failure
`erealloc(...)`	`realloc( ...)`	Exits on failure

Table A.1: Summary of *eprintf* functions

A.2.2 Using *asserts*

My view is that *asserts* should not be used as a lazy programmer's way of informing end-users of predictable error conditions in the operation of a program. Rather, they should be used mainly during the development stage to help the programmer figure out where things are going wrong.

Despite this, the source code often uses asserts in this "lazy" way.

A.2.3 Incorrect conventions

I use one coding "standard" that is incorrect and may lead to portability problems. In particular, there are occasions where the following assumptions are made:

1. A generic pointer `void *` is the same size as an integer (`int`) and data of one type can be cast to the other.

2. A generic pointer `void *` and a pointer to a function `void *()(...)` are the same size and either can be cast to the other.

Both of these assumptions *violate* the formal specifications in the ANSI C standard. They are, however, very commonly encountered.

Using these conventions makes some of the code easier to write and more readable. Note that there is NO assumption about the size of these things. Normally, however, they are all either 16 or 32 bits.

Some of the problems explore ways to avoid these assumptions.

A.2.4 Miscellaneous conventions

Some of the source code follows some other arbitrary conventions that are a matter of personal choice. These include:

Addresses of functions: If `funcP` is a function pointer data type and `foo()` is a function, I use `funcP = &foo` to set `funcP` to be a pointer to the function `foo()`. Other programmers use the shorter and equivalent form: `funcP = foo`.

Private header files: If a header file is used *only* by the programmer implementing a module and does not need to be viewed or included by programmers using the module, I append the letter 'P' to the name of the header. For example, `foo.h` would be a public header file, while `fooP.h` would be a private (non-distributed) header file.

private and public: In implementing modules that have "static globals" (i.e. declarations made at the highest level in the source code file but qualified as "static" to prevent their names being exported to the linker), I often use the pre-processor statements:

```
#define private static
#define public
```

This allows me to declare things at the global level as "private" or "public" which I find closer to the semantics I have in mind.

Underscores for private names: When a name (such as a `typedef` or a private variable is used in a module, I usually prepend an underscore character ('_') to its "logical" name.

A.3 Conventions used in preparing this book

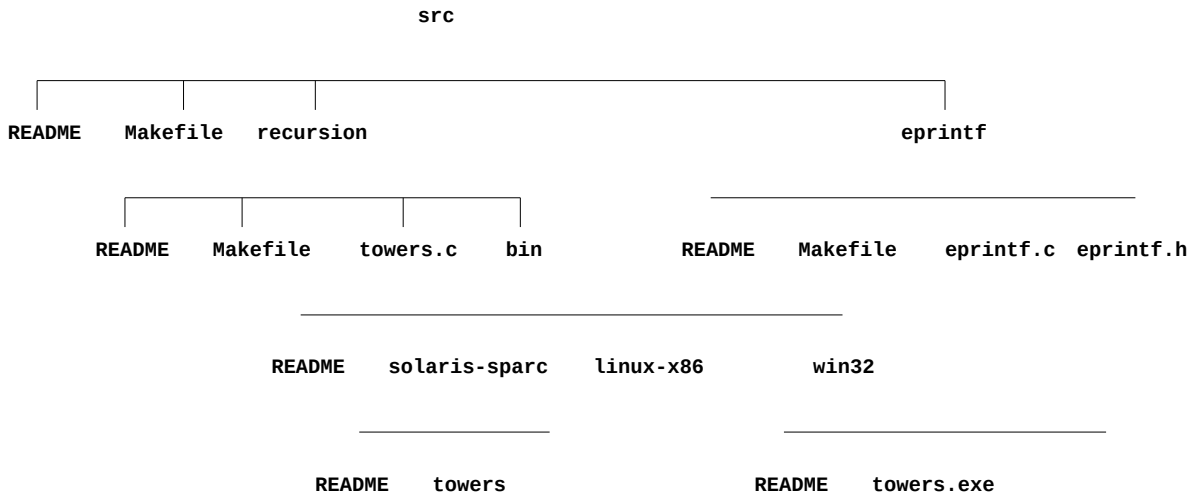


Figure A.1: The source tree organization for this book

Appendix B

Data Structures, Memory and Pointers

This appendix is a review of the important C programming topics involving data structures, memory usage (especially dynamic allocation and freeing), and pointers. Because it is a review, it is terse. (Some of the problems—and their answers—give more details.)

B.1 Data structures

A *data structure* is a collection of related data items that can be manipulated as a single entity. As a simple example, suppose you wish to treat a person's age (in years) and height (in centimeters) as a single item. We could define a data structure type as follows:

```
typedef struct AgeHeight AgeHeight;
struct AgeHeight {
    unsigned int age;     /* years */
    unsigned int height;  /* centimetres */
};
```

These statements define the new data type `AgeHeight`, but they do not allocate any storage or create any variables of this type. To do so, you declare variables of this data type as follows:

```
AgeHeight jane, dick, sally, spot;
AgeHeight friends[10];
```

Note that we can also initialize the values of fields in a structure with the following syntax:

```
 /* Tom is 8 years old and 1.3m tall */
AgeHeight tom = {8, 130};
```

Here we have defined four simple variables of this data type (`jane`, etc.) and an array—`friends`—of `AgeHeight` data structures. We can access fields (i.e. the component parts) with the "dot"(.) syntax as follows:

```
/* On Jane's birthday... */
jane.age++;
```

The data structure as a whole can also be manipulated; it can be used in argument lists, can be returned, its address can be evaluated, it can be assigned, and the `sizeof` operator used on it. For example:

```
AgeHeight OlderAndTaller(AgeHeight ah, int grew)
{
   ah.age++;
   ah.height += grew;
   return ah;
}

/* The Birthday Girl grew more than an inch last year */
jane = OlderAndTaller(jane, 3);
```

If we determined the size (amount of memory) the data structure uses with `sizeof jane` (or `sizeof AgeHeight`) we would get 8 bytes—the size of 2 `int`s[1]

B.1.1 Strings in data structures

This porridge is too hot.
This porridge is too cold.
This porridge is just right!

—Goldilocks

[1] In these examples, we assume that the size of an `int` or pointer is 32 bits, the default sizes on most popular computer systems. Systems with 16-bit or 64-bit architecture may use different sizes, but the general principles about data structure size remain valid.

We now consider a different data structure that raises new design and implementation issues. We wish to treat a person's name as a data structure containing two fields: first name and last name. The design issue we face is whether to implement the data structure as two arrays of characters (`Name_arr`) or as two pointers to null-terminated strings (`Name_str`). The two different ways of doing this are shown below:

```
#define MAX 32 /* max num. of chars in first or last name */
typedef struct Name_arr Name;
//or typedef struct Name_str Name;
struct Name_arr {
    char first[MAX];
    char last[MAX];
};

struct Name_str {
   char * first;
   char * last;
};
```

In the first array-based implementation, all of the information is in the data structure itself; in the `char *` implementation, the actual strings are located elsewhere in memory and the data structure itself only contains references to them.

The array-based version is conceptually simpler, but comes at the price of less flexibility and (usually) more space requirements. Both of these limitations come about because the array of characters has a fixed size. If this fixed size is too small, the data structure will be unable to handle long names; if it is too big, more memory is wasted. (Only Goldilocks knows the size that is "just right".)

In the example above, the maximum number of characters in both the first and last name fields is fixed at 32 each; hence the data structure occupies 64 bytes. The size of the data structure is always 64 bytes which is wasteful for short names like "Jane Doe". In the rare case when both arrays are completely filled (each name is 31 characters with the last array slot used for the null string terminator), memory is used as efficiently as possible.

The alternate pointer based structure always requires 8 bytes of memory for the two pointers and additional storage elsewhere for the actual strings.

Unlike, the array-based approach, there is no built-in limit to the size of the strings and only the minimum amount of memory is needed.

The clear advantages of the pointer-based approach does come at a price, however. Since the memory needed for the strings is not part of the data structure itself, it has to be allocated separately. Even worse, if the data structure has only limited lifetime (for example, if it is a local variable) the memory allocated for the strings has to be explicitly freed to avoid "memory leaks".[2]

To see how the use of the array-based approach is easier than the pointer one, suppose we wanted to read a last name from `stdin`. With the `Name_arr` structure, we could write:

```
struct Name_arr name;
scanf("%s", &name.last);
```

It is quite a bit more complicated when the `Name_str` data structure is used:

```
char buf[MAX_LEN];
Name_str name;
int len;

/* read stdin into a temporary buffer */
scanf("%s", buf);
/* Determine how many characters were read */
len = strlen(buf);
/* Allocate memory for the chars PLUS null terminator */
name.last = malloc(len+1);
/* Copy the buffer to the allocated memory */
strcpy(name.last, buf);
```

Sometimes, however, the `Name_str` is easier to use than the `Name_arr` version. This is especially true when a string for either the first or last name fields already exists. For example, compare:

[2]The "memory leak" problem is far from trivial to avoid in large software projects. All too often it is unclear when and where dynamically allocated memory should be released. In some programming languages—lisp and Java for example—the problem is avoided with a technique called *garbage collection* that periodically finds chunks of memory that are no longer being used and frees them. Indeed, this is one of the attractive features of Java that have helped make it so popular.

```
struct Name_arr dick;
strcpy(dick.last, "Tricky");
```

with:

```
struct Name_str dick;
dick.last = "Tricky";
```

B.1.2 Compound data structures

Note that a field within a data structure can itself be a structure and so on recursively. For example, the following definition of a "Person" data type combines the previous data structures and auxiliary information:

```
typedef struct AgeHeight AgeHeight;
struct AgeHeight {
    unsigned int age;     /* years */
    unsigned int height;  /* centimetres */
};

typedef struct Name_arr Name;
struct Name_arr {
    char first[MAX];
    char last[MAX];
};

typedef struct Person Person;
struct Person {
     char gender;
     AgeHeight ageHt;
     Name name;
}
```

A `Person` data type could then be used as follows:

```
Person tom, dick, harry;
tom.gender = 'M';
harry.ageHt.age = 20;
strcpy(dick.name.first, "Dick");
```

B.2 Pointers to data structures

Love and marriage, love and marriage
Go together like a horse and carriage

—J. Van Heusen/S. Cahn

Pointers and data structures also go together "like a horse and carriage". Indeed, there is special notation for accessing the field of a structure from a pointer to it as shown below:

```
AgeHeight ah, *ah_p;
ah_p = &ah;   /* Initialize pointer */
ah.age = 8;
ah_p->height = 120; /* Use '->' to access field via pointer */
```

It is very common, for example, to pass a parameter as a pointer to a data structure rather than the structure itself. This is both more efficient and more flexible. It is more efficient because a pointer to any structure, no matter how large, is always four bytes. It is more flexible because the called function can access and modify the actual fields of the data structure. We could re-write the previous function `OlderAndTaller` more efficiently as:

```
void OlderAndTallerP(AgeHeight * ahp, int grew)
{
   ahp->age++;
   ahp->height += grew;
}

OlderAndTallerP(&jane, 3);
```

Sometimes you want to get the efficiency of passing a pointer to a structure, but you do want the called function to be able to modify the structure. (The called function could not modify the structure if the whole thing were passed since this gives access only to a *copy* of the structure and any changes to it have no effect on the original.) To achieve this, simply declare the argument as a `const`. Not only will the compiler disallow modifications inside the called function, it may also be able to optimize the running time of the program more effectively.

For example:

```
unsigned int TotalHeight(const AgeHeight * a1,
                         const AgeHeight * a2)
{
    return a1->height + a2->height;
}
```

B.2.1 Linked structures

Many engineering concepts and designs can be visualized with diagrams containing boxes and interconnections of various sorts. Well known examples include state-machine diagrams, control system diagrams, flowcharts, block diagrams, etc. Almost all of these kinds of visualizations can be converted into a set of data structures that contains all the information in the diagram.

We illustrate the general technique with a simple example: our solar system and galaxy. Of course, we could draw a picture like Figure B.1.

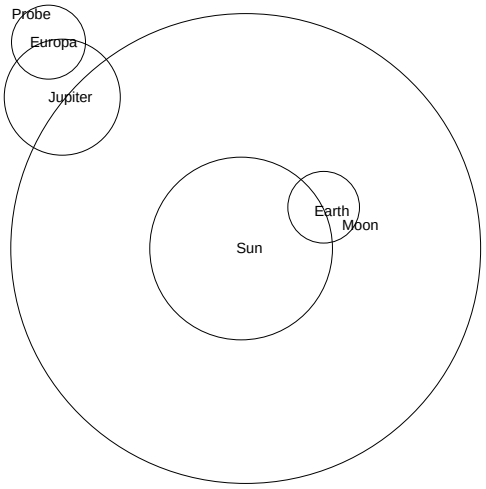


Figure B.1: Our Solar System (partial, not to scale)

This picture corresponds (very roughly) to the physical structure of part of our solar system. But it does not highlight what we are really interested in: what body orbits which other body and so on. The picture would get quite messy, for example, if we tried to include the Sun orbiting the black hole at the center of our galaxy and a (hypothetical) NASA probe orbiting

Jupiter's moon Europa. But we can easily distill this information using a more abstract diagram shown in Figure B.2.

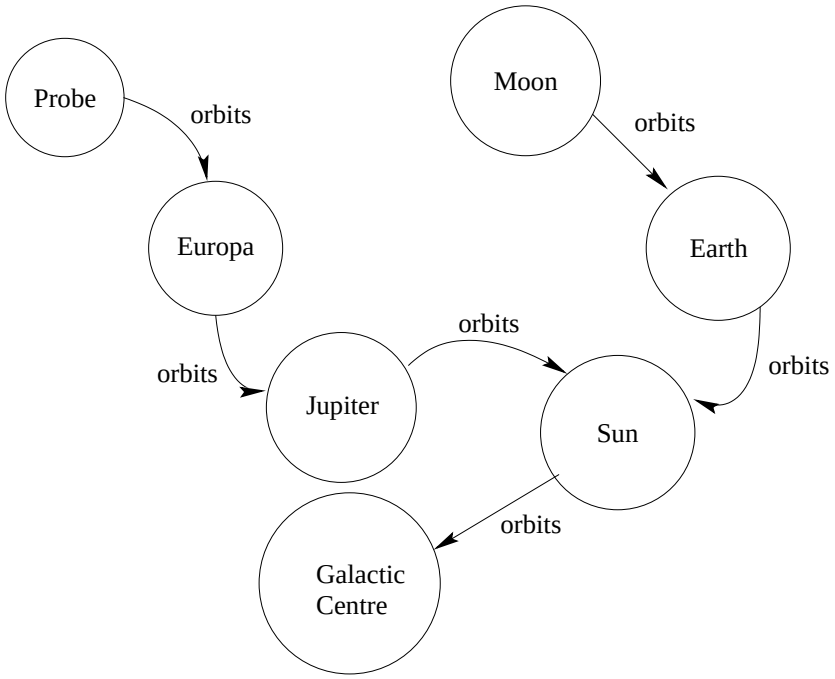


Figure B.2: Some Celestial Bodies

To convert information in this figure into a set of data structures, we first note that each circle contains two pieces of information: the name of the celestial body (Earth, Moon, etc) and an arrow labeled "orbits" that points to the body it orbits. We conclude that each circle corresponds to a data structure with two fields: `name` and `orbits`. We will call the data structure type `Body`, use `char *` data type for `name` and a pointer to a Body structure as the data type for `orbits`. Note also that we have defined a "helper" data type—`BodyP`—to indicate a pointer to a `Body`.

The type definitions are as follows:

```
typedef struct Body Body, * BodyP;
struct Body {
   char * name;
   BodyP orbits;
```

```
};
```

Next we declare the data types of variables corresponding to each circle:

```
Body probe, jupiter, earth, moon, sol,
     europa, galacticCentre;
```

Finally, we define the values of the data structures:

```
Body probe          = {"Probe", &europa};
Body jupiter        = {"Jupiter", &sol};
Body earth          = {"Earth", &sol};
Body moon           = {"Moon", &earth};
Body sol            = {"Sun", &galacticCentre};
Body europa         = {"Europa", &jupiter};
Body galacticCentre = {"Black hole", NULL};
```

We can now write a simple function that prints out the sequence of orbits. (For example, if the function is called with the argument `probe`, it would print: "Probe orbits Europa orbits Jupiter orbits Sun orbits Black hole orbits nothing.")

```
void orbits(BodyP b)
{
   assert(b != NULL);
   do {
      printf("%s orbits ", b->name);
      b = b->oribts;
   } while (b != NULL);
   printf("nothing.\n");
}
```

Complete working code based on the examples in this appendix can be found in Appendix E or in the directory `src/dataStructsAndPtrs`.

B.3 Problems

B.1 Why do I use the idiom:

```
typedef struct Foo Foo;
struct Foo {
   .
   .
};
Foo foo;
```

instead of:

```
struct Foo {
   .
   .
};
struct Foo foo;
```

or:

```
typedef struct {
   .
   .
} Foo;
Foo foo;
```

B.2 Add Mars and its two moons to the diagram and to the initialization of the data structs in C.

B.3 Suppose we wished to put information explicitly into our celestial bodies diagram (Figure B.2) that shows what is orbited by a body. (For example, the sun is orbited by the Earth and Jupiter.)

1. How would you modify the diagram to show this information?
2. Modify the `Body` data structure to reflect the new information.
3. What kind of general data structure have you now built (considering only the new links you have added).?

B.4 What will the following print:

```
printf("%s\n", probe->orbits->orbits->name);
```

B.5 Many programmers use the typedef:

```
typedef char * String;
```

Discuss the pros and cons of this kind of `typedef`.

B.6 Write a program that reads zero or more lines of text in the following format:

```
<lastName> <firstName> <gender> <age> <height>
```

A set of `Person` type records should be initialized using this information.

B.7 As written, the `orbits()` function produces results like:
"Europa orbits Jupiter orbits Sun orbits Black hole orbits nothing."
How can the function and data structures be modified so that common words such as "sun" or "moon" be preceded with the article "the"? Indicate also how the word "the" should be expressed as "The" or "the"... (although we do not want to get into bizarre arguments about what the word "the" means despite some Presidential musings...)

Appendix C

Modules, Linking and Scope

This appendix is a review of the important topics in the organization of source code into separate files and modules.

C.1 A Simple Example

C.2 Problems

Appendix D

Solutions

The solutions to most of the problems are given in this Appendix.

Please inform me of any errors (it would be virtually impossible that I have not made some) you find. I would also be interested in alternative solutions.

D.1 Answers for Chapter 1

1.1 Step 1 Set the *minimum value* to UNDEFINED (whose "numerical" value is assumed to be ∞).

Step 2 Examine the next card (if there is one). If it is less than *minimum value*, change the *minimum value* to this card's value.

Step 3 If there are no more cards, the answer is *minimum value* and STOP.

Step 4 Go back to Step 2.

1.2

FindDuplicates Algorithm

Find any duplicates in n *cards*

Step 1: Sort the cards with an $n \log n$ sort algorithm like MergeSort.

Step 2: Go through the cards one by one remebering the last card examined. If there are any duplicates, the last and current cards will be the same. If this happens, STOP and report the duplicates. If the end of the deck is reached, STOP and report "no duplicates".

Step 1 is of $n \log n$ complexity. Step 2 is clearly of linear complexity and is swamped by Step 1 for large n; hence the entire algorithm is of $n \log n$ complexity.

1.3 If the card is in the deck, we can choose any one at random and, if we are very lucky, it will be "the" card. So, it *is possible* to find a particular number by looking at only one card.

A simple way to find a card would be to look at each card in the deck one after the other. If you find the card you are looking for, great. If you don't find it before reaching the end of the deck, it is not there. This *linear* method will require, on average, $n/2$ operations to find a card; you need to look at all n of them to be sure that the card you are looking for is not there. (This method does not take advantage of the deck being in sorted order; it is simple to implement and can be used with any pile of cards whether or not they are in sorted order.)

More seriously, look first at the middle card. If its value is what we are looking for, STOP. Otherwise, if its value is bigger than the one we are looking for, search for it among the 511 sorted cards smaller than the middle one. If the search value is bigger, search in the other 511 cards. In either case, the problem is has now been reduced to searching amongst 511 cards after only one comparison. Repeating the method again reduces the search space to 255 cards.

If the card exists, we will find with no more than 9 comparisons. If the card is not in the deck, we need 10 comparisons.

1.4

Tables D.2 and D.1 show the manual calculations for two cases including the most general case.

n	$T(n/2)$	$2T(n/2)+n$	$n\lg n+n$
2	1	$2\times 1+2=4$	4
4	4	$2\times 4+4=12$	12
8	12	$2\times 12+8=32$	32
16	32	$2\times 32+16=80$	80
32	80	$2\times 80+32=192$	192
64	192	$2\times 192+64=448$	448

Table D.1: $2T(n/2)+n$ for selected values assuming $T(1)=1$

n	$T(n/2)$	$2T(n/2)+bn$	$n\lg n$
2	a	$2a+2b$	2
4	2(a+b)	$2\times 2(a+b)+4b=4a+8b$	8
8	$4(a+2b)$	$2\times(4a+8b)+8b=8a+24b$	24
16	$8a+24b$	$2\times(8a+24b)+16b=16a+64b$	64
32	$16a+64b$	$2\times(16a+64b)+32b=32a+160b$	160
64	$32a+160b$	$2\times(32a+160b)+64b=64a+384b$	384

Table D.2: $2T(n/2)+bn$ for selected values where T(1) = a

For the most general case, we guess that the closed form solution is $T(n)=bn\lg n+an$. This is clearly true for $n=1$. We now use mathematical

induction to prove that it is true for T(2n):

$$\begin{aligned} T(2n) &= 2T(n) + 2n \\ &= 2bn \lg n + 2an + 2bn \\ &= 2nb(\lg n + lg2) + 2an \\ &= b \times 2n \lg 2n + a \times 2n \end{aligned}$$

which completes the proof.

1.5

CalculateAverage Algorithm

Calculate and output the average of n *numbers (the input)*

Step 1: Set *average* $\leftarrow$ UNDEFINED.

Step 2: Set *total* $\leftarrow 0$.

Step 3: Set $i \leftarrow 0$

Step 4: If there are no more numbers to read, set *average* $\leftarrow$ *total*/i if $i \neq 0$. Output *average* (the answer) and STOP.

Step 5: Read the next input number and add it to *total*.

Step 6: Set $i \leftarrow i + 1$

Step 7: Go back to *Step 4*.

Note that the way this algorithm handles the case of zero items differs from how it was done in the CalculateTotal algorithm. In calculating the total, the answer was defined as 0 if there were no inputs. When calculating the average, however, we use the special value UNDEFINED as the answer when there is nothing to average. Other versions of the algorithm could use a different convention (including the same convention we used for CalculateTotal.) The convention used, however, should be clearly stated to the user of the algorithm.

The convention used here would be more difficult to implement in C as we could no longer use the primitive data type `int` as the return value since

the special "value" UNDEFINED would have to differ from all `int`s; we would have to use some type of data structure or pointer (using a NULL pointer to represent UNDEFINED) as the return value.[1]

1.6 This solution is only available to professors.

1.7 Not yet available.

1.8 The number of steps is $2+3n$. The time is $T(n) = T_1+T_2+n(T_3+T_4+T_2)$, or, $T(n) = c_1 n + c_0$ where c_1 is $T_2 + T_3 + T_4$ and c_0 is $T_1 + T_2$. It is *linear*.

1.9

linear: 25 seconds (because increasing the size of a linear problem by a factor of 5, increases the time by the same factor.)

logarithmic: 5.8 seconds (because $T(n) = K \log n$, hence $K = 5/(\log 20000) = 5/4.3 = 1.16$ and $T(100000) = K \log 100000 = 5.8$.)

cubic: 625 seconds (because increasing the size of a cubic problem by a factor of 5, increases the time by a factor of $5^3 = 125$.)

$n \log n$: 29 seconds (by combining the linear factor—5—with the logarithmic factor—1.16.)

quadratic: 125 seconds (because increasing the size of a quadratic problem by a factor of 5, increases the time by a factor of $5^2 = 25$.

constant: 5 seconds (independent of problem size)

1.10 Tell me your results.

1.11 Logarithms of one base can be converted to another base by multiplication by a constant. In particular:

$$\log_a x = \frac{\log_b x}{\log_b a}$$

Hence, if an algorithm is logarithmic to base a, we can say:

$$T(n) = k \log_a n$$

[1] We could, however, use IEEE floating point numbers, returning the value `NaN` ("not a number") for the case of zero inputs.

But this can be written as:

$$\begin{aligned} T(n) &= k \log_b n / \log_b a \\ &= \frac{k}{\log_b a} \log_b n \\ &= k' \log_b n \end{aligned}$$

In other words, changing the base only changes the multiplication constant.

1.12

SelectionSortList Algorithm

Sort a list of n *elements*

Step 1: Set *head* $\leftarrow$ beginning of list.

Step 2: If *head* is NIL, STOP.

Step 3: Set *small* $\leftarrow$ smallest item on the list beginning with *head.*

Step 4: Interchange the data at *head* and *small* unless either is NIL.

Step 5: Set *head* to next item on list.

Step 6: Go back to *Step 2*

1.13

a) No.

b) Yes.

c)
```
void easter(int Y)
  {
    int G, C, X, Z, D, E, N;

    G = Y%19 + 1;
```

```
  C = Y/100 + 1;
  X = (3*C)/4 - 12;
  Z = (8*C + 5)/25 - 5;
  D = 5*Y/4 - X - 10;
  E = (11*G + 20 + Z - X)%30;
 /* Following needed because C's mod operator incorrect. */
  if (E < 0)
    E += 30;
  if (((E==25) && (G > 11)) || (E == 24))
    E++;
  N = 44 - E;
  if (N < 21)
    N += 30;
  N = N + 7 - (D+N)%7;
  if (N > 31) {
    printf("%d April %d\n", N-31, Y);
  } else {
    printf("%d March %d\n", N, Y);
  }
  return;
}
```

d) I regularly use the *emacs* text editor; it has a command—*holidays*—which tells me (amongst other things) the date for Easter. A portion of the *elisp* source code is:

```
(defun holiday-easter-etc ()
 "List of dates related to Easter, as visible in calendar window."
 (let* ((century (1+ (/ displayed-year 100)))
  (shifted-epact        ;; Age of moon for April 5...
   (% (+ 14 (* 11 (% displayed-year 19));;...by Nicaean rule
      (-     ;;...corrected for the Gregorian century rule
      (/ (* 3 century) 4))
      (/    ;;...corrected for Metonic cycle inaccuracy.
      (+ 5 (* 8 century)) 25)
      (* 30 century));;              Keeps value positive.
      30))
 (adjusted-epact     ;;  Adjust for 29.5 day month.
```

```
  (if (or (= shifted-epact 0)
      (and (= shifted-epact 1)
           (< 10 (% displayed-year 19))))
       (1+ shifted-epact)
    shifted-epact))
  paschal-moon ;;Day after the full moon on/after March 21.
 - (calendar-absolute-from-gregorian
       (list 4 19 displayed-year))
      adjusted-epact))
.
.
.
  (list (calendar-gregorian-from-absolute
            (- abs-easter 49))
          "Shrove Sunday")
.
.
.
   (list (calendar-gregorian-from-absolute
             (+ abs-easter 60))
           "Corpus Christi")))
```

e) Because the modern Gregorian calendar was established in that year following a papal bull, *Inter Gravissimas*, issued by Pope Gregory XIII on February 24, 1582. The rules for calculating Easter changed. (For the fanatically curious, the Latin original and an English translation are available at `http://www.bluewaterarts.com/calendar/InterGravissimas.htm`)

f) No and no.

1.14 Not yet available.

1.15 Not yet available.

1.16 Not yet available.

1.17 The only changes required are the declarations of `array[]` and `tmp` to be doubles instead of ints:

```
void mySort(double array[], unsigned int first, unsigned int last)
{
  int i;
  /* Step 1: Is there nothing to sort? */
  while (first < last)
     /* Step 2: Swap... */
     for(i = first+1; i <= last; i++) {
       /* Find smallest one in rest of array */
       if(array[first] >  array[i])) {
         /Step 2..continued...swap them */
         double tmp;
         tmp = array[first]
         array[first] = array[i];
         array[i] = tmp;
      }
     first++;
  }
  return;
```

1.18 The declarations of `array[]` and `tmp` must be changed to be `char *` instead of ints. Also, since strings (unlike ints or doubles) cannot be compared with the `>` operator, we need to use the standard `strcmp` function:

```
void mySort(char * array[], unsigned int first, unsigned int last)
{
  int i;
  /* Step 1: Is there nothing to sort? */
  while (first < last)
     /* Step 2: Swap... */
     for(i = first+1; i <= last; i++) {
       /* Find smallest one in rest of array */
       if(strcmp(array[first],array[i]) > 0) {
         /Step 2..continued...swap them */
         char * tmp;
         tmp = array[first]
         array[first] = array[i];
         array[i] = tmp;
      }
```

```
    first++;
  }
  return;
```

1.19 Not yet available.

1.20 Not yet available.

1.21 Not yet available.

1.22 1. $T(n) = 1.01^{n-1}\mu$sec.

2. $T(100) \approx 2.7\mu$sec, $T(3000) \approx 9.2 \times 10^6$ sec ≈ 3 months.

D.2 Answers for Chapter 2

2.1
```
int nWaysToMakeChange(int amount, int nCoinTypes, int coinsUsed[])
{
  int used2[5];

  if (amount == 0) {
    showCoinsUsed(coinsUsed);
    return 1;
  }
  if (nCoinTypes == 1) {
    coinsUsed[0] = amount;
    showCoinsUsed(coinsUsed);
    return 1;
  }

  if (amount < 0)
    return 0;
  memmove(used2, coinsUsed, 5*sizeof(int));
  used2[nCoinTypes-1]++;
  return nWaysToMakeChange(amount, nCoinTypes-1, coinsUsed)
    + nWaysToMakeChange(amount - typesOfCoins[nCoinTypes-1],
                        nCoinTypes, used2);
}
```

2.2 It will still work. There are fewer steps if the pink number is the smallest.

2.3 The recursion goes on forever. The algorithm can be fixed by incrementing the "pink" number and decrementing the "blue" number when the "pink" number is negative.

2.4 Solving the problem requires 18,446,744,073,709,551,615 moves or 584.5 billion years if they are moved at the rate of one per second. Since the universe is thought to be no older than 20 billion years, there is nothing to worry about (yet!). Alas, if the moves are executed at the rate of one per millisecond, the universe should have been destroyed about 580 million years after the Big Bang.

Suppose, however, that the Big Bang occurred *exactly* 15 billion years ago (from the time you start reading this answer) and the moves are made at the rate of one ever 25.86 milliseconds then Watch Out! (You may not be able to read to the end of this sentence.)

2.5 The C program is:

```
#include <assert.h>
#include <stdio.h>
#include <stdlib.h>

unsigned int gcd(unsigned int m, unsigned int n)
{
  unsigned int r;

  r = m%n;
  if (r == 0)
    return n;
  else
    return gcd(n, r);
}

int main(int argc, char * argv[])
{
  unsigned int n1, n2;
  assert(argc == 3);
  n1 = atoi(argv[1]);
```

```
  n2 = atoi(argv[2]);
  printf("GCD of %d and %d is %d\n", n1, n2, gcd(n1, n2));
  exit(0);
}
```

2.6 Not yet available

2.7

```
#include <stdio.h>
#include <assert.h>
#include <stdlib.h>
int fib(int);

int main(int argc, char * argv[])
{
  int n;
  n = atoi(argv[1]);
  printf("Fib{%d): %d\n", n, fib(n));
  exit(0);
}

int fib(int n)
{
  unsigned int fibNumbers[20];
  int i;

  assert(n >= 1 && n < 20);
  fibNumbers[0] = 1;
  fibNumbers[1] = 1;
  for(i = 2; i < n; i++)
    fibNumbers[i] = fibNumbers[i-1] + fibNumbers[i-2];
  return fibNumbers[n-1];
}
```

2.8 Not yet available

2.9 Not yet available

2.10 We "guess" that the closed form solution is: $M(n) = 2^n - 1$. This is clearly true for the base case ($n = 0$).

We prove it is true in general using mathematical induction where we assume it is true for n and prove that this implies it is true for $n + 1$:

$$\begin{aligned} M(n+1) &= 2M(n) + 1 \text{ (by definition)} \\ &= 2(2^n - 1) + 1 \text{ (the hypothesis)} \\ &= 2^{n+1} - 2 + 1 \\ &= 2^{n+1} - 1 \end{aligned}$$

proving the hypothesis.

2.11 Not yet available

2.12
```
#include <stdio.h>
#include <stdlib.h>

static char * digits = "0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ";

void printVal(unsigned int i, unsigned int base)
{
  if (i < base) {
    putchar(digits[i]);
  } else {
    printVal(i/base, base);
    putchar(digits[i%base]);
  }
}

int main(int argc, char * argv[])
{
  int n, b;
  if(argc != 3) {
    fprintf(stderr, "Usage: %s number base\n", argv[0]);
```

```
    exit(2);
  }
  n = atoi(argv[1]);
  b = atoi(argv[2]);
  printVal(n, b);
  putchar('\n');
  exit(0);
}
```

2.13 Not yet available.

2.14 Not yet available

2.15 Not yet available

2.16 Not yet available

2.17

$$\text{mul}(m,n) = \begin{cases} 0 & \text{if } n = 0 \\ \text{mul}(m, n-1) + m & \text{if } n > 0 \end{cases}$$

2.18 Not yet available.

2.19 Not yet available.

2.20 $Gib(n) = (k + c)Fib(n) - k$

2.21 The results (in seconds) for calculating $Fib(n)$ for selected values using `fib.c` (the recursive version) and `fib-linear.c` are shown in the table below. (Note that we stop at $n = 47$ because $Fib(47)$ is the biggest Fibonacci number that fits into a 32-bit binary number.)

While there is no perceptible difference in performance for small numbers, the superiority of the linear algorithm is striking for larger numbers.

n	Linear	Recursive
20	0.0	0.0
25	0.0	0.0
30	0.0	0.2
35	0.0	1.9
40	0.0	21.7
45	0.0	241.2
47	0.0	634.9

Table D.3: Times for calculating $Fib(n)$ with linear and exponential algorithms

D.3 Answers for Chapter 3

3.1 An interpreter.

3.2 Compilation.

3.3 `<while_statement> ::= 'while' '(' <expression> ')' <statement>`

3.4
```
      <expr> ::= <term> { <addop> <term> }
<term> ::= <factor> { <mulop> <factor> }
>factor> ::= <base> { '^' <base> }
<base> ::= <num>
             | '(' <expr> ')'
             | '-' <expr>
<addop> ::= '+' | '-'
<mulop> ::= '*' | '/'
```

3.5 Not yet available.

3.6 Not yet available.

3.7 Not yet available.

3.8 The keywords in C are: `auto, break, case, char, const, continue, default, do, double, else, enum, extern, float, for, goto, if, int, long, register, return, short, signed, sizeof, static, struct, switch, typedef, union, unsigned, void, volatile, while`.

The separators and operators, are: `[ ] ( ) { } . -> ++ -- ~ ! - + & * / % << >> < > <= >= == != ^ | && || ? : = += -= *= /= %= <<= >>= &= ^= |= ,`

3.9 Pure lisp has the fewest number of syntactical rules (parentheses must balance). C++ inherits all of C's syntax and then some... It is the most complex language that I regularly program in.

3.10

3.11

3.12 Not yet available.

3.13 Not yet available.

3.14 Not yet available.

3.15 Not yet available.

3.16 The only function that needs to change is `factor()`:

```
/** factor() parses and evaluates a <factor> as defined
 *  by the BNF:
 *  <factor> ::= <num> | '(' <expr> ')'
 *
 * ENTRY CONDITIONS: The next unprocessed token must
 *                   be in the token global variable.
 * EXIT CONDITIONS:  The next unprocessed token will be
 *                   in the token global variable.
 *
 * @return The value of the factor.
 */
int factor(void)
{
  int value;

  if (isdigit(token)) {
    value = token - '0';
    token = getNextToken();
    return value;
  }else if (token == '-') {
    token = getNextToken();
```

```
    value = expr();
    return -value;
  } else {
    assert(token == '(');
    token = getNextToken();
    value = expr();
    assert(token == ')');
    token = getNextToken();
  }
  return value;
}
```

3.17

```
<simple_sentence> ::= <noun_phrase> <verb> <noun_phrase>
<noun_phrase> ::= <article> { <adjective> } <noun>
<article> ::= 'the'
              | 'a'
<adjective> ::= 'hungry'
                | 'black'
                | 'white'
                | 'cute'
<noun> ::= 'cat' | 'dog'

<verb> ::= 'chased' | 'ate' | 'likes'
```

D.4 Answers for Chapter 4

4.1

1. The Θ complexities are:

   (a)

   $$n^2 + n + 5 = \Theta(n^2)$$

   (b)

   $$200n + 6 = \Theta(n)$$

(c)

$$\lg n! = \Theta(n \log n)$$

2. If $n = 100$, the running times are:

(a)

$$n^2 + n + 5 = 100^2 + 100 + 5 = 10105$$

(b)

$$200n + 6 = 20006$$

(c)

$$\lg n! = 524.7649933$$

4.2 First, $T(n) = \Theta(n^3)$. (Note that $T(n) = 2.6n^3 + \lg n^{10} + 123.456 = 2.6n^3 + 10 \lg n + 123.456$; hence the cubic term is the fastest growing.)

This implies it is $O(n^3)$ and $\Omega(n^3)$. It is also $O()$ of any function that grows faster than a cubic one and $\Omega()$ of anything that grows more slowly than a cubic function.

Thus:

1. $T(n)$ is $O(n^6)$ is true.
2. $T(n)$ is $\Omega(n^6)$ is false.
3. $T(n)$ is $\Theta(n^6)$ is false.
4. $T(n)$ is $O(n^3)$ is true.
5. $T(n)$ is $\Omega(n^3)$ is true.
6. $T(n)$ is $\Theta(n^3)$ is true.
7. $T(n)$ is $O(n^2)$ is false.
8. $T(n)$ is $\Omega(n^2)$ is true.
9. $T(n)$ is $\Theta(n^2)$ is false.

4.3 One way to prove this is to show:

1. $\sum_{1 \leq i \leq n} i^k = O(n^{k+1})$
2. $\sum_{1 \leq i \leq n} i^k = \Omega(n^{k+1})$

First:

$$\sum_{1 \leq i \leq n} i^k < \sum_{1 \leq n} n^k = n^{k+1}$$

Hence, $\sum_{1 \leq i \leq n} i^k = O(n^{k+1})$.

Second:

$$\sum_{1 \leq i \leq n} i^k > \sum_{n/2 \leq i \leq n} (n/2)^k \geq (1/2)(n/2)^{k+1} \geq K n^{k+1}$$

Hence, $\sum_{1 \leq i \leq n} i^k = \Omega(n^{k+1})$.

4.4 An informal solution:
Note that $K_1 n! + P_k(n) = \Theta(n!)$
Hence, $\log(K_1 n! + P_k(n)) = \log \Theta(n!) = \Theta(n \log n)$

4.5 The `for` loop implies that f takes the values:
$c_1, c_1 c_2, c_1 c_2^2, c_1 c_2^3 \cdots$
Hence, f will get bigger if and only if $c_2 > 1$ and, in this case, it will grow exponentially.
More specifically, the number of times the loop will execute is:

$$\lceil \log_{c_2}(n/c_1) \rceil + 1$$

Hence, the loop has $\Theta(\log n)$ complexity.
The loop will terminate *only if* the series:

$$c_1, c_1 c_2, c_1 c_2^2, c_1 c_2^3 \cdots$$

increases.
Hence, we require that $c_1 c_2 > 1$

n	Derivation	$T(n)$
1	$T(1) = c$	c
2	$T(2) = 2T(1) + 2a + b = 2c + b + 2a$	$2a + b + 2c$
4	$T(4) = 2T(2) + 4a + b = (4a + 2b + 4c) + 4a + b$	$8a + 3b + 4c$
8	$T(8) = 2T(4) + 8a + b = (16a + 6b + 8c) + 8a + b$	$24a + 7b + 8c$

Table D.4: $T(n) = 2T(n/2) + an + b$ for selected values

4.6 Given:

$$T(n) = \begin{cases} c & \text{if } n = 1 \\ 2T(n/2) + an + b & \text{otherwise} \end{cases}$$

We start with the manually derived Table D.4

Next, we draw the recurrence tree for the general case as shown in Figure D.1.

We can now see the closed form solution for $T(n)$:

$$T(n) = na \lg n + (n-1)b + nc$$

Clearly, then $T(n) = \Theta(n \lg n)$ (because $na \lg n$ is the fastest growing term in the closed-from solution.)

4.7 This is equivalent to showing that n^ϵ is $\Omega(\lg n)$.

We know that for all n:

$$T(2n) = \lg 2n = \lg n + 1 = T(n) + 1$$

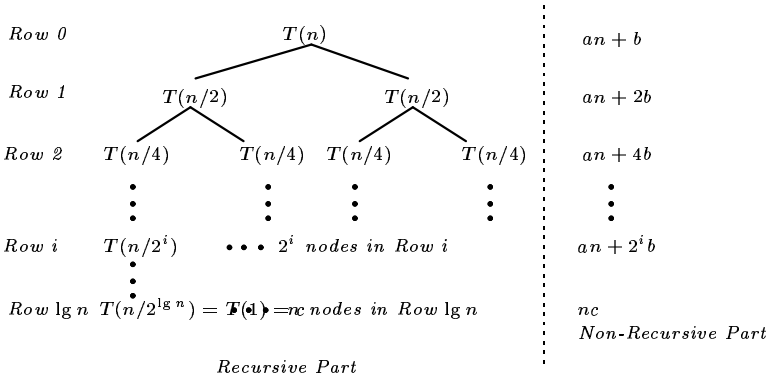
So we need only show that:

$$T(2n) - T(n) = (2n)^\epsilon - n^\epsilon > 1$$

for sufficiently large $n > n_0$.

We have:

$$\begin{aligned} (2n)^\epsilon - n^\epsilon &> 1 \\ n^\epsilon(2^\epsilon - 1) &> 1 \\ n &> \left(\frac{1}{2^\epsilon - 1}\right)^{1/\epsilon} \end{aligned}$$



Summary:
Row 0 contribution: $an + b$
Row 1 contribution: $an + 2b$
Row 2 contribution: $an + 4b$
Row i contribution: $an + 2^i b$
Row $\lg n$ *contribution:* nc
Total: $(n \lg n)a + (n-1)b + nc$

Figure D.1: Tree for $T(n) = 2T(n/2) + an + b$

D.5 Answers for Chapter 5

5.1 The possible sets are `{}` (the empty set), `{1}`, `{2}` and `{1 2}` for a total of 4. (Note that `{2 1}` is not a separate set since the order of elements is immaterial.)

There are an infinite number of allowable bags; obviously, all the sets are permissible, but so are bags like `{1 1 1 1 }` (a bag with four 1's in it).

5.2 Not yet available.

5.3 Not yet available.

5.4 Not yet available.

5.5

Not yet available.

5.6

```
int myGetSize(IntLLBag b)
{
  int s;
  IntLLBag b2;
  b2 = newIntLLBag();
  for(s = 0; !isEmpty(b); s++)
      addIntLLBag(b2, removeIntLLBag(b));
  while!isEmpty(b2))
  addIntLLBag(b, removeIntLLBag(b2));
  destroyIntLLBag(b2);
  return s;
}
```

5.7 Not yet available.

5.8 Not yet available.

5.9 Not yet available.

D.6 Answers for Chapter 6

6.1 The code is:

```
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>

/*** Typedefs ***/
typedef enum{Penny = 1,
             Nickel = 5,
             Dime = 10,
             Quarter = 25,
             HalfDollar = 50}
        Coin;

typedef struct
{
  int amt;
```

```
  unsigned int nTypes;
}  Params;

static unsigned int top = 0;
#define STACK_SIZE 1000
static Params stack[STACK_SIZE];



/*** Globals ***/
Coin typesOfCoins[] = {Penny, Nickel, Dime, Quarter, HalfDollar};

/*** Function prototypes ***/
int nWaysToMakeChange(int amount, int nCoinTypes);

static void push(Params p)
{

  stack[top++] = p;
  return;
}

static void initStack()
{
  top = 0;
  return;
}

static Params pop()
{
  return stack[--top];
}

static int isEmptyStack()
{
  return (top == 0);
}
```

```
int main(int argc, char * argv[])
{
  if ((argc != 2) || (atoi(argv[1]) < 0)) {
    fprintf(stderr, "Usage: %s amount(in cents)\n", argv[0]);
    exit(1);
  }
  printf("%d\n", nWaysToMakeChange(atoi(argv[1]),
   sizeof(typesOfCoins)/sizeof(int)));
  exit(0);
}

int nWaysToMakeChange(int amount, int nCoinTypes)
{
  int ans = 0;

  Params p;
  p.amt = amount;
  p.nTypes = nCoinTypes;
  initStack();
  push(p);
  while(!isEmptyStack()) {
    p = pop();
    if ((p.amt == 0) || (p.nTypes == 1)) {
      ans++;
      continue;
    }
    if (p.amt < 0)
      continue;
    p.nTypes--;
    push(p);
    p.amt -= typesOfCoins[p.nTypes];
    p.nTypes++;
    push(p);
  }
  return ans;
}
```

6.2

6.3
```
typedef int StackData;   /* For example */
static unsigned int top = 0;
#define STACK_SIZE 1000
static StackData stack[STACK_SIZE];

static void push(StackData p)
{
  stack[top++] = p;
  return;
}

static void initStack()
{
  top = 0;
  return;
}

static StackData pop()
{
  return stack[--top];
}

static int isEmptyStack()
{
  return (top == 0);
}

static DataType set(int index, DataType data)
{
  DataType old;
  assert(index >= 0 && index < top);
  old = stack[index];
  stack[index] = data;
  return old;
}
```

```
static unsigned int getSize(void)
{
  return top;
}

static DataType get(int index)
{
  assert(index >= 0 && index < top);
  return stack[index];
}
```

6.4 Not yet available.

6.5 Not yet available.

6.6
```
top--->   local variable x (in foo)
          local variable y (in foo)
          local variable z (in foo)
          return address main_2
          formal parameter p = 5
          formal parameter q = 6
```

6.7 Because it is extremely poor practice to use any part of memory that has been released. While it is highly probable that the re-written code will work and be slightly more efficient, there is a possibility that it will fail, especially in a multi-threaded system with a pre-emptive scheduler and lots of interrupts.

6.8 Not yet available.

6.9 Not yet available.

6.10 Not yet available.

6.11 Not yet available.

6.12 All the operations are $\Theta(1)$.

D.7 Answers for Chapter 7

7.1 Not yet available.

7.2 Not yet available.

7.3 The original tree looks is shown in Figure D.2.

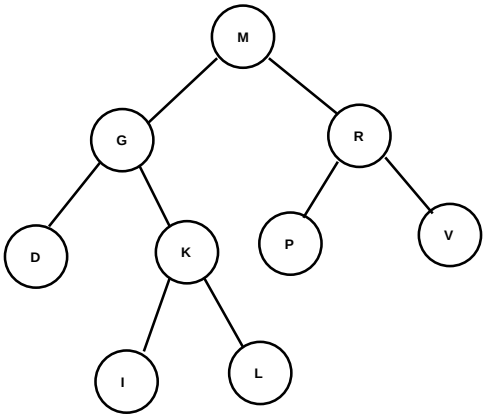


Figure D.2: A simple binary tree

The traversal orders are:

inorder `D, G, I, K, L, M, P, R, V`

pre-order `M, G, D, K, I, L, R, P, V`

post-order `D, I, L, K, G, P, V, R, M`

Figure D.3 shows the tree after deleting the G node.
It is described in parenthetical notation as:

```
(M (I (D) (K () (L))) (R (P) (V)))
```

7.4 Note yet available.

7.5 Not yet available.

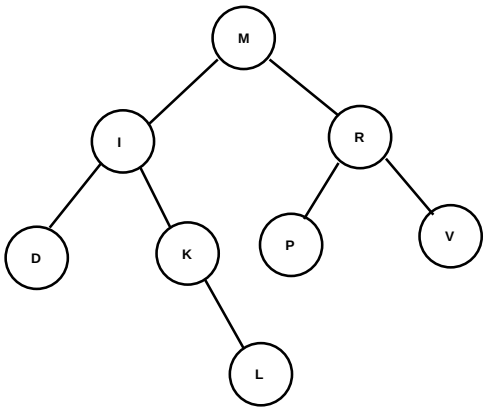


Figure D.3: A simple binary tree with G node deleted

D.8 Answers for Chapter 8

8.1 a. 2.

b. 4.

c. Yes as shown in Figure D.4

8.2 The evolution is shown in Figure D.5

D.9 Answers for Chapter 9

9.1 Figure D.6 shows the hash tables.

D.10 Answers for Chapter 10

10.1 1. The graph is shown in Figure D.7

D.11 Answers for Appendix B

B.1 Not yet available.

B.2 We would add:

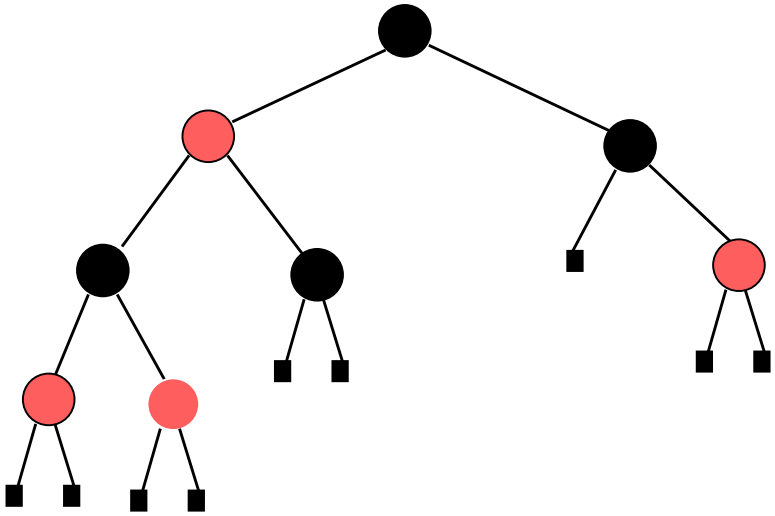


Figure D.4: Colored tree

```
Body mars, deimos, phobos ;
Body mars    = {"Mars", &sol};
Body deimos  = {"Deimos", &mars};
Body phobos  = {"Phobos", &mars};
```

B.3 Not yet available.

B.4 "Jupiter".

B.5 Not yet available.

B.6 Not yet available.

B.7 Not yet available.

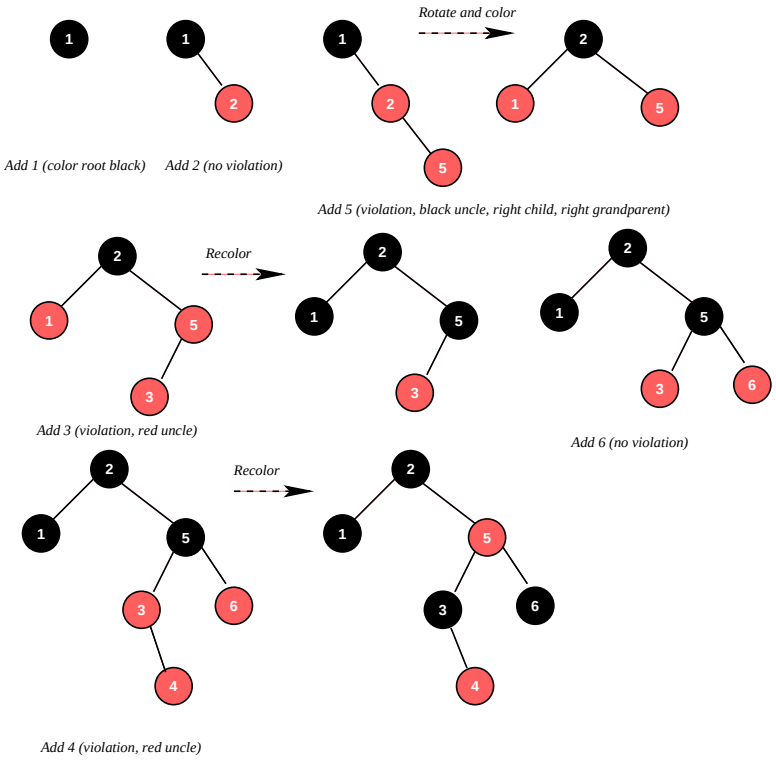


Figure D.5: Building a Red-Black Tree

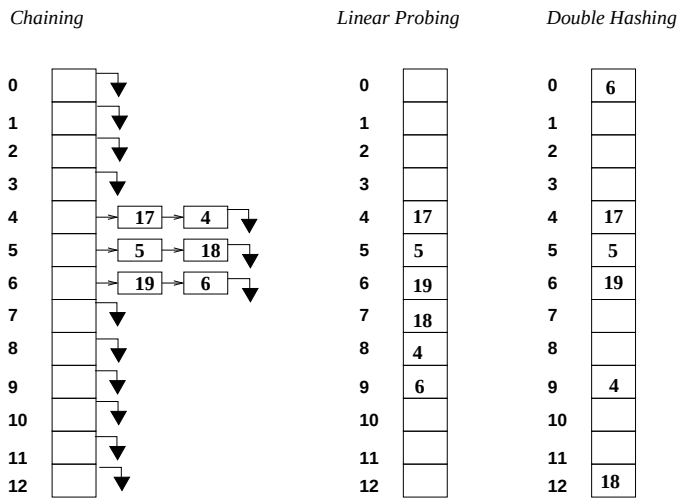


Figure D.6: Resulting Hash tables after adding "5, 19, 17, 18, 4, 6"

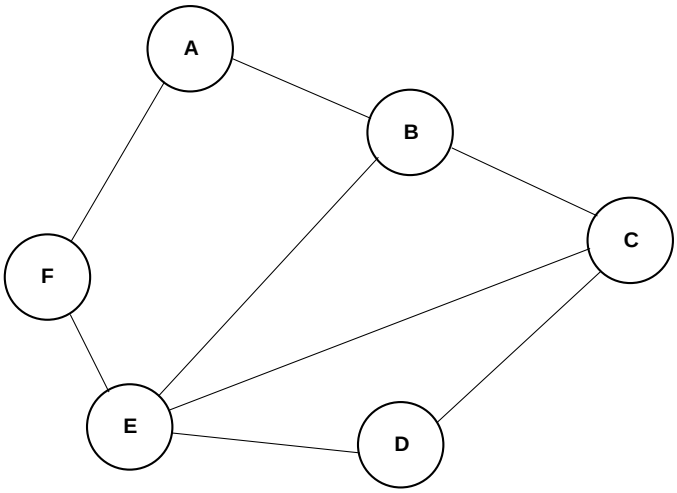


Figure D.7: Resulting Graph

Appendix E

Source code

This appendix includes all of the source code discussed in the text (including the problems).

Note that all of the code does *not* respect the basic coding standards given in A. There are also inconsistencies.(I'm working on it...)

E.1 Algorithms

Source code for Chapter 1. This source code can also be found in the directory `src/algorithms`.

E.1.1 README

```
This directory contains the source code for programs
discussed in Chapter 1---Algorithms of the book
"Engineering Algorithms and Data Structures".

Makefile  --- the Makefile (what else)

metrics.c --- the implementation of the "metrics" package.
metrics.h --- the header file for the "metrics" package.
selectionSort.c --- recursive implementation of Selection Sort
sortDriver.c --- a driver for any implementation of "mySwap"
```

E.1.2 Makefile

```
CFLAGS=-Wall -g -ansi -pedantic
EXECS= selSort easter testMetrics
CC=gcc

all: ${EXECS}

selSort: selectionSort.o sortDriver.o metrics.o eprintf.o
gcc -o selSort selectionSort.o sortDriver.o metrics.o eprintf.o

easter: easter.o eprintf.o
gcc -o easter easter.o eprintf.o


testMetrics: metrics.c
gcc -o testMetrics -DTEST_METRICS metrics.c

clean:
-@rm -f *~ *.dvi *.log *.ps *.log *.aux *.o ${EXECS} junk*
```

E.1.3 metrics.h

```
/* Copyright (C) 1999 Ken Clowes (kclowes@ee.ryerson.ca) */

#ifndef METRICS_H
#define METRICS_H

int myCompare(int, int);
void mySwap(int *, int *);
void myCopy(const int *, int *);
unsigned int getNumCompares();
unsigned int getNumCopies();
unsigned int getNumSwaps();
```

```
#endif /* #ifndef METRICS_H */
```

E.1.4 metrics.c

```
/* Copyright (C) 1999 Ken Clowes (kclowes@ee.ryerson.ca) */

/**
 * The metrics module provides utility functions to compare, swap
 * and copy ints.  The module tracks the number of times each operation
 * is performed and provides functions to obtain these numbers.
 * The primary use of the metrics module is to facilitate the
 * performance measurement and comparison of sorting algorithms.
 *
 * The module also includes a stand alone self-test main routine
 * allowing the automatic testing of its components.  The test executable
 * may be obtained by compiling this module with TEST_METRICS #defined.
 */
static unsigned int numCompares = 0;
static unsigned int numSwaps = 0;
static unsigned int numCopies = 0;

/**
 * myCompare compares two ints. The function returns an integer
 * greater than, equal to, or less than 0, if the first number
 * is greater than, equal to, or less than the second number
 * respectively.
 *
 * @param n1 The first number.
 * @param n2 The second number.
 */
int myCompare(int n1, int n2)
{
  numCompares++;
  return (n1 - n2);
}

/**
```

```
 * mySwap interchanges two ints.
 *
 * Example:
 *    int a[] = {1, 2, 3, 4};
 *    mySwap(&a[0], &a[3]);
 *    printf("a[0] = %d; a[3] = %d\n", a[0], a[3]);
 * will produce the output:
 *    a[0] = 4; a[3] = 1
 *
 * @param ip1 A pointer to the first int.
 * @param ip2 A pointer to the second int.
 */
void mySwap(int * ip1, int * ip2)
{
  int tmp;
  numSwaps++;
  tmp = *ip1;
  *ip1 = *ip2;
  *ip2 = tmp;
  return;
}

/**
 * myCopy copies an int elsewhere.
 * @param ip1 A pointer to the int that is to be copied
 * @param ip2 A pointer of where to copy the first int to.
 */

void myCopy(const int * ip1, int * ip2)
{
  numCopies++;
  *ip2 = *ip1;
  return;
}

/**
 * getNumCompares returns the number of times myCompare
 * was invoked.
```

```
 */
unsigned int getNumCompares()
{
  return numCompares;
}

/**
 * getNumCopies returns the number of times myCopy
 * was invoked.
 */
unsigned int getNumCopies()
{
  return numCopies;
}

/**
 * getNumSwaps returns the number of times mySwap
 * was invoked.
 */
unsigned int getNumSwaps()
{
  return numSwaps;
}

#ifdef TEST_METRICS
#include <stdio.h>
#include <assert.h>
int main()
{
  int data[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
  int nTests = 0;
  int tmp1, tmp2;
  /* Ensure that the initial counts are all zero. */
  assert(getNumSwaps() == 0); nTests++;
  assert(getNumCompares() == 0); nTests++;
  assert(getNumCopies() == 0); nTests++;

  /* Test compare operations */
```

```
  assert(myCompare(5, 5) == 0); nTests++;
  assert(myCompare(12, 34) < 0); nTests++;
  assert(myCompare(-5, -10) > 0); nTests++;
  assert(myCompare(0xffffffff, 0) < 0); nTests++;
  assert(getNumCompares() == 4); nTests++;

  /* Test swap operations */
  tmp1 = data[2];
  tmp2 = data[3];
  assert(tmp1 != tmp2);
  mySwap(&data[2], &data[3]);
  assert((tmp1 == data[3]) && (tmp2 == data[2])); nTests++;
  mySwap(&data[2], &data[3]);
  assert((tmp1 == data[2]) && (tmp2 == data[3])); nTests++;
  assert(getNumSwaps() == 2); nTests++;

  /* Test copy operations */
  myCopy(data+5, &tmp1);
  assert(tmp1 == 6); nTests++;
  assert(tmp1 != tmp2);
  myCopy(&tmp1, &tmp2);
  assert(tmp2 == 6); nTests++;
  assert(getNumCopies() == 2); nTests++;

  printf("The metrics module passed all %d tests\n", nTests);
  exit(0);
}
#endif /* TEST_METRICS */
```

E.1.5 selectionSort.c

```
#include "metrics.h"

/** mySort sorts a sub-array of int's.  The array name and the
 * starting and ending indices are passed as parameters.  The function
 * modifies the array such that all of its elements between the start
 * and end indices (inclusively) are in sorted order from smallest to
 * largest.
```

```
 *
 * Example:
 *  If you write:
 *      int data[] = {-5, 20, -17, 63, 6};
 *      mySort(data, 0, 4);
 *      for(int i = 0; i <= 4; i++)
 *         printf("%d\n", data[i]);
 *
 * the following will be printed to stdout:
 *     -17
 *     -5
 *     6
 *     20
 *     63
 *
 * @param array  The array to be modified.
 * @param first Index of first element of the sub-array.
 * @param last Index of  last element of the sub-array.

 *
 */
void mySort(int array[], unsigned int first, unsigned int last)
{
  int i;
  /*
   * This version of "mySort" uses the Selection Sort algorithm
   * as described in class:
   *   Step 1: If there are no cards to sort, then STOP.
   *   Step 2: Otherwise, find the smallest card,
   *      remove it and place it on top of the sorted card pile.
   *   Step 3: Go back to step 1.
   *
   * To implement the basic idea of the algorithm in the context
   * of sorting a sub-array of ints instead of a deck of cards,
   * we first re-phrase the algorithm as follows:
   *
   * Step 1: If the sub-array to be sorted contains no elements
   *   (i.e. if the first index is >= the last index), then
```

```
 *   simply return.
 *
 * Step 2: Otherwise, replace the array element "array[first]"
 *    with the smallest int in the sub-array "array[first]..array[last]"
 *    and ensure that the original value of array[first] is somewhere
 *    in the smaller sub-array "array[first+1]..array[last]" whenever
 *    array[first] is modified.
 *
 * Step 3: Solve the smaller problem of sorting the sub-array
 *    "array[first+1]..array[last]"
 *
 *  (NOTE: The original algorithm has simple recusive and
 *  iterative implementations; we have chosen a recursive
 *  implementation.)
 *
 * The C implementation of the array-based algorithm follows.
 */

/* Step 1: Is there nothing to sort? */
if (first >= last)
  return;

/* Step 2: Make array[first]  the minimum in array[first]..array[last].
           Modify array[first] only with "swap" operations. */
for(i = first+1; i <= last; i++) {
  if(myCompare(array[first], array[i]) > 0) {
    mySwap(&array[first], &array[i]);
  }
}

/* Step 3 */
mySort(array, first+1, last);

return;
}
```

E.1.6 sortDriver.c

```
/* Copyright (C) 1999 Ken Clowes (kclowes@ee.ryerson.ca) */

#include <stdio.h>

#define MAX_SIZE 100000
#include "metrics.h"
#include "eprintf.h"
extern void mySort(int array[], unsigned int first, unsigned int last);

int main(int argc, char * argv[])
{
  int a[MAX_SIZE];
  unsigned int array_size, i;

  setprogname(argv[0]);
  if (argc != 1) {
    eprintf("with NO additional command line args\n");
    exit(1);
  }

  /* Read ints from stdin into an array */
  for(array_size = 0; (scanf("%d", &a[array_size]) != EOF)
        && (array_size < MAX_SIZE);
      array_size++)
    ;

  /* sort the array */
  if(array_size > 0) {
    mySort(a, 0, array_size-1);
  }

  /* Print out the modified array */
  for(i = 0; i < array_size; i++)
    printf("%d\n", a[i]);

  /* Print stats */
```

```
  fprintf(stderr, "Comparisons: %d\n", getNumCompares());
  fprintf(stderr, "Swaps: %d\n", getNumSwaps());
  fprintf(stderr, "Copy operations: %d\n", getNumCopies());

  exit(0);
}
```

E.1.7 easter.c

```
/** Determine the Date of Easter for any year after 1582.
 *
 *  Invoked from the command line as:
 *       easter <Year>
 *
 *  Example:
 *       easter 2009
 *    produces (on stdout)
 *       12 April 2009
 */

#include <stdio.h>
#include <stdlib.h>
#include "eprintf.h"

void easter(int year);

int main(int argc, char * argv[])
{
  int Y;
  setprogname(argv[0]);
  if (argc != 2) {
    eprintf("Usage: easter year (where year > 1582)");
  }
  Y = atoi(argv[1]);
  if (Y < 1582) {
    eprintf("Year must be greater than 1582");
  }
  easter(Y);
```

```
  exit(0);
}

/*
 * The algorithm is implemented using the description
 * from Knuth's "The Art of Computer Programming", Vol 1.
 * page 159--160.
 */

void easter(int Y)
{
  int G, C, X, Z, D, E, N;

  G = Y%19 + 1;
  C = Y/100 + 1;
  X = (3*C)/4 - 12;
  Z = (8*C + 5)/25 - 5;
  D = 5*Y/4 - X - 10;
  E = (11*G + 20 + Z - X)%30;
 /* Following needed because C's mod operator incorrect. */
  if (E < 0)
    E += 30;
  if (((E==25) && (G > 11)) || (E == 24))
    E++;
  N = 44 - E;
  if (N < 21)
    N += 30;
  N = N + 7 - (D+N)%7;
  if (N > 31) {
    printf("%d April %d\n", N-31, Y);
  } else {
    printf("%d March %d\n", N, Y);
  }
  return;
}
```

E.2 Recursion

Source code for Chapter 2. This source code can also be found in the directory `src/recursion`.

E.2.1 README

```
This directory contains the files associated with
Chapter 2---Recursion---in the book "Engineering
Algorithms and Data Structures".

CountChange.c  -- Count ways to make change using
                  1c, 5c,10c, 25c and 50c coins
CountChangeShowWays.c -- Show how each way is formed
Makefile -- the makefile (what else)
euclid.c -- Euclid's algorithm for gcd
fib-linear.c -- A linear complexity Fibonacci generator
fib.c -- Recursive Fibonacci
goodTowers.c -- Towers of Hanoi
towers.c -- Towers of Hanoi (buggy version)
pv.c -- print value to any base
```

E.2.2 CountChange.c

```
#include <stdio.h>
#include <stdlib.h>

/*** Typedefs ***/
typedef enum{Penny = 1,
             Nickel = 5,
             Dime = 10,
             Quarter = 25,
             HalfDollar = 50}
        Coin;

/*** Globals ***/
Coin typesOfCoins[] = {Penny, Nickel, Dime, Quarter, HalfDollar};
```

```
/*** Function prototypes ***/
int nWaysToMakeChange(int amount, int nCoinTypes);

int main(int argc, char * argv[])
{
  if ((argc != 2) || (atoi(argv[1]) < 0)) {
    fprintf(stderr, "Usage: %s amount(in cents)\n", argv[0]);
    exit(1);
  }
  printf("%d\n", nWaysToMakeChange(atoi(argv[1]),
   sizeof(typesOfCoins)/sizeof(int)));
  exit(0);
}

int nWaysToMakeChange(int amount, int nCoinTypes)
{
  if ((amount == 0) || (nCoinTypes == 1))
    return 1;
  if (amount < 0)
    return 0;
  return nWaysToMakeChange(amount, nCoinTypes-1)
    + nWaysToMakeChange(amount - typesOfCoins[nCoinTypes-1], nCoinTypes);
}
```

E.2.3 CountChangeShowWays.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

/*** Typedefs ***/
typedef enum{Penny = 1,
             Nickel = 5,
             Dime = 10,
             Quarter = 25,
             HalfDollar = 50}
        Coin;
```

```
/*** Globals ***/
Coin typesOfCoins[] = {Penny, Nickel, Dime, Quarter, HalfDollar};

char * coinName[] = {"Penn", "Nickel", "Dime", "Quarter", "HalfDollar"};
/*** Function prototypes ***/
int nWaysToMakeChange(int amount, int nCoinTypes, int coinsUsed[]);
void showCoinsUsed(int coinsUsed[]);

int main(int argc, char * argv[])
{
  int coinsUsed[] = {0, 0, 0, 0, 0};
  if ((argc != 2) || (atoi(argv[1]) < 0)) {
    fprintf(stderr, "Usage: %s amount(in cents)\n", argv[0]);
    exit(1);
  }
  nWaysToMakeChange(atoi(argv[1]), sizeof(typesOfCoins)/sizeof(int),
                    coinsUsed);
  exit(0);
}

int nWaysToMakeChange(int amount, int nCoinTypes, int coinsUsed[])
{
  int used2[5];

  if (amount == 0) {
    showCoinsUsed(coinsUsed);
    return 1;
  }
  if (nCoinTypes == 1) {
    coinsUsed[0] = amount;
    showCoinsUsed(coinsUsed);
    return 1;
  }

  if (amount < 0)
    return 0;
  memmove(used2, coinsUsed, 5*sizeof(int));
  used2[nCoinTypes-1]++;
```

```
  return nWaysToMakeChange(amount, nCoinTypes-1, coinsUsed)
    + nWaysToMakeChange(amount - typesOfCoins[nCoinTypes-1],
                        nCoinTypes, used2);
}

void showCoinsUsed(int coinsUsed[])
{
  int i;
  int n = sizeof(typesOfCoins)/sizeof(int);
  int printed = 0;

  for(i = 0; i < n; i++) {
    if (coinsUsed[i] != 0) {
      if (printed)
        printf(" and ");
      printf("%d %s", coinsUsed[i], coinName[i]);
      if ( i == 0 ) {
        printf("%s", (coinsUsed[i] == 1) ? "y" : "ies");
      } else if (coinsUsed[i] > 1)
        printf("s");
      printed = 1;
    }
  }
  printf("\n");
  return;
}
```

E.2.4 Makefile

```
CFLAGS=-Wall -g
EXECS=towers fib CountChange fib-linear euclid \
      CountChangeShowWays pv goodTowers
CC=gcc

all: ${EXECS}

towers: towers.o
gcc -o towers towers.c
```

```
goodTowers: goodTowers.o eprintf.o
gcc -o goodTowers goodTowers.o eprintf.o

fib: fib.o
gcc -o fib fib.o

fib-linear: fib-linear.o
gcc -o fib-linear fib-linear.o

euclid: euclid.o
gcc -o euclid euclid.o

CountChange: CountChange.o
gcc -o CountChange CountChange.o

CountChangeShowWays: CountChangeShowWays.o
gcc -o CountChangeShowWays CountChangeShowWays.o

pv: pv.o
gcc -o pv pv.o



clean:
-@rm -f *~ *.dvi *.log *.ps *.log *.aux *.o ${EXECS} *.exe
```

E.2.5 euclid.c

```
#include <assert.h>
#include <stdio.h>
#include <stdlib.h>

unsigned int gcd(unsigned int m, unsigned int n)
{
  unsigned int r;
```

```
  r = m%n;
  if (r == 0)
    return n;
  else
    return gcd(n, r);
}

int main(int argc, char * argv[])
{
  unsigned int n1, n2;
  assert(argc == 3);
  n1 = atoi(argv[1]);
  n2 = atoi(argv[2]);
  printf("GCD of %d and %d is %d\n", n1, n2, gcd(n1, n2));
  exit(0);
}
```

E.2.6 fib-linear.c

```
#include <stdio.h>
#include <assert.h>
#include <stdlib.h>
int fib(int);

int main(int argc, char * argv[])
{
  int n;
  n = atoi(argv[1]);
  printf("Fib{%d): %d\n", n, fib(n));
  exit(0);
}

int fib(int n)
{
  unsigned int fibNumbers[200];
  int i;

  assert(n >= 1 && n < 200);
```

```
  fibNumbers[0] = 1;
  fibNumbers[1] = 1;
  for(i = 2; i < n; i++)
    fibNumbers[i] = fibNumbers[i-1] + fibNumbers[i-2];
  return fibNumbers[n-1];
}
```

E.2.7 fib.c

```
#include <stdio.h>
#include <assert.h>
#include <stdlib.h>
int fib(int);

int main(int argc, char * argv[])
{
  int n;
  n = atoi(argv[1]);
  printf("Fib{%d): %d\n", n, fib(n));
  exit(0);
}

int fib(int n)
{
  assert(n >= 1);
  if((n==1) || (n==2))
    return 1;
  return fib(n-1) + fib(n-2);
}
```

E.2.8 goodTowers.c

```
/* Copyright (c) 1999 Jane Smith (jsmith@ee.ryerson.ca) */

/** The functions in this file solve the classic Towers of Hanoi
 * problem.
 */
```

```
/*** Includes ***/
#include <stdlib.h>
#include <stdio.h>
#include <assert.h>

/*** Function prototypes ***/
void towers(int n, int from, int to);

/** * main manages the command line interface to solving the
 *    Towers of Hanoi problem.  The command line args (which
 *    must be string representations of numbers) indicate the
 *    number of disks to be moved and the source and
 *    destination tower numbers.  (The towers are identified
 *    with the numbers 1, 2 and 3.)
 *      @param argc the number of command line arguments
 *      @param argv a pointer to an array of strings where:
 *        There must be exactly 3 arguments where:
 *         - the fist arg is the number of disks to move
 *         - the second is the ID-number of the source
 *         - the third is the ID of the destination
 *      @return returns an exit code of 0 at
 *         completion unless invoked incorrectly or if system
 *         resources or user patience is exceeded;
 *         in those cases an exit code of 1 (for invokation
 *         errors) or some non-zero value (for impatience
 *         or resource exhaustion) is returned.
 */
int main(int argc, char * argv[])
 {
   int nDisks, sourceTower, destTower;
   if (argc != 4) {
     eprintf(stderr, "Usage: towers n_disks source dest\n");
   }
   nDisks = atoi(argv[1]);
   if(nDisks < 0) {
     eprintf("Number of disks must be non-negative\n");
   }
   sourceTower = atoi(argv[2]);
```

```
    if(sourceTower < 1 || sourceTower > 3) {
      eprintf("Source tower ID must be 1, 2 or 3\n");
    }
    destTower = atoi(argv[3]);
    if(destTower < 1 || destTower > 3) {
      eprintf("Destination tower must be 1, 2 or 3\n");
    }
    if(destTower == sourceTower) {
      eprintf("Source and Destination must be different\n");
    }

    towers(nDisks, sourceTower, destTower);
    exit(0);
  }

/**
  * "towers" solves the Towers of Hanoi problem
  * and writes the solution to <stdout> as 2^n -1
  * lines in the form:
  *       <from> <to>
  *  where:
  *     <from> is the ID of a tower to pick up a disk from
  *     <to>   is the ID of where to drop the disk to
  * @param n     the number of disks to move
  * @param from  the tower ID number to move from
  * @param to    the tower ID number of the destination
  */
void towers(int n, int from, int to)
     /*
      * The standard recursive "divide and conquer" method
      * ois used to solve the problem.  Specifically:
      *     1) If the number of disks  is 0, then STOP.
      *     2) Otherwise, move n-1 disks to the spare tower.
      *     3) Move a single disk to the destination.
      *     4) Move n-1 disks from the spare to destination
      */
{
  if( n > 0) {
```

```
    /* Note: "spare", "from" and "to" are distinct
     * and chosen from 1 or 2 or 3.  Hence, we must have
     * the invariant:
     *     spare + from + to = 1 + 2 + 3 = 6
     */
    int spare = 6 - from - to;
    --n;
    towers(n, from, spare);
    printf("%d %d\n", from, to);
    towers(n, spare, to);
  }
}
```

E.2.9 towers.c

```
/* Copyright (c) 1999 Jane Smith (jsmith@ee.ryerson.ca) */

/** The functions in this file solve the classic Towers of Hanoi
 * problem.
 */

/*** Includes ***/
#include <stdlib.h>
#include <stdio.h>
#include <assert.h>

/*** Function prototypes ***/
void towers(int n, int from, int to);

/** * main manages the command line interface to solving the
 *    Towers of Hanoi problem.  The command line args (which
 *    must be string representations of numbers) indicate the
 *    number of disks to be moved and the source and
 *    destination tower numbers.  (The towers are identified
 *    with the numbers 1, 2 and 3.)
 *        @param argc the number of command line arguments
 *        @param argv a pointer to an array of strings:
 *            There must be exactly 3 arguments:
```

```
 *             -- the fist arg is the number of disks
 *             -- the 2nd is the ID of the source tower
 *             -- the 3rd arg is the ID of the destination
 *        @return returns an exit code of 0 at
 *           completion unless invoked incorrectly
 *           or if system resources or user patience
 *           is exceeded; in those cases an exit code
 *           of 1 (for invokation errors) or some non-zero
 *           value (for impatience or resource exhaustion)
 *           is returned.
 */
int main(int argc, char * argv[])
 {
   int nDisks, sourceTower, destTower;
   if (argc != 4) {
     fprintf(stderr, "Usage: towers num_disks source dest\n");
     exit(1);
   }
   nDisks = atoi(argv[1]);
   if(nDisks < 0) {
     fprintf(stderr, "Number of disks to move must be non-negative\n");
     exit(1);
   }
   sourceTower = atoi(argv[2]);
   if(sourceTower < 1 || sourceTower > 3) {
     fprintf(stderr, "Source tower number must be 1, 2, or 3\n");
     exit(1);
   }
   destTower = atoi(argv[3]);
   if(destTower < 1 || destTower > 3) {
     fprintf(stderr, "Destination tower must be 1, 2, or 3\n");
     exit(1);
   }
   if(destTower == sourceTower) {
     fprintf(stderr, "Source and Destination towers must be different\n");
     exit(1);
   }
```

```
   towers(nDisks, sourceTower, destTower);
   exit(0);
 }

/**
  * "towers" solves the Towers of Hanoi problem and writes the solution
  * to <stdout> as 2^n -1 lines in the form:
  *       <from> <to>
  *  where:
  *     <from> is the ID of a tower to pick up a disk from
  *     <to>   is the ID of where to drop the disk to
  * @param n     the number of disks to move
  * @param from  the tower ID number to move from
  * @param to    the tower ID number of the destination
  */
void towers(int n, int from, int to)
     /*
      * The standard recursive "divide and conquer" method is used
      * to solve the problem.  Specifically:
      *     1) If the number of disks to move is 0 (zero), then STOP.
      *     2) Otherwise, move n-1 disks to the spare tower.
      *     3) Move a single disk to the destination.
      *     4) Move n-1 disks from the spare tower to the destination
      */
{
  if( n > 0) {
    /* Note: "spare", "from" and "to" are distinct and chosen from
     * 1 or 2 or 3.  Hence, we must have the invariant:
     *     spare + from + to = 1 + 2 + 3 = 6
     */
    int spare = 6 - from - to;
    --n;
    towers(n, from, spare);
    printf("%d %d\n", from, to);
    towers(n, to, spare);
  }
}
```

E.3 Parsing

Source code for Chapter 3. This source code can also be found in the directory `src/parsing`.

E.3.1 README

```
This directory contains the files associated with
Chapter 3---Parsing---in the book "Engineering
Algorithms and Data Structures".

Makefile -- the makefile (what else)
calc.c -- a simple calcultor (algebraic expressions)
nounPhraseWordCounter.c -- simple BNF example
```

E.3.2 Makefile

```
CFLAGS=-Wall -g -ansi -pedantic
EXECS=nounPhraseWordCounter calc calc2
CC=gcc

all: ${EXECS}

calc: calc.o
gcc -o calc calc.o

calc2: calc2.o
gcc -o calc2 calc2.o

nounPhraseWordCounter: nounPhraseWordCounter.o
gcc -o nounPhraseWordCounter nounPhraseWordCounter.o

clean:
-@rm -f *~  *.dvi *.log *.ps *.log *.aux *.o \
         *.eps  core junk* *.bak ${EXECS}
```

E.3.3 calc.c

```
/* Copyright (C) 1999 Ken Clowes (kclowes@ee.ryerson.ca) */

/** This program parses and interprets arithmetic expressions and
 * prints their values.  Arithmetic expressions are defined by the BNF:
 *
 * <pre>
 *    <expr> ::= <term> { <addop> <term> }
 *    <term> ::= <factor> { <mulop> <factor> }
 *    <factor> ::= <num>
 *                 | '(' <expr> ')'
 *    <addop> ::= '+' | '-'
 *    <mulop> ::= '*' | '/'
 * </pre>
 *
 * This program is pedagogical; it shows you how to write an interpreter
 * using "recursive descent parser" techniques for a recursive BNF.
 * However, it is (obviously) not meant to be a real calculator since
 * its tokenizer can only recognize single-digit numbers.  Nonetheless,
 * it is not difficult to modify the tokenizer for more meaningful
 * multi-digit representations of numbers.  We leave this as an
 * exercise.
 *
 * The exit code is zero unless a syntax error is detected.
 */

/* ** System includes ** */
#include <stdio.h>
#include <ctype.h>
#include <assert.h>

/* ** typedefs ** */
typedef int token_t;

/* ** globals ** */
token_t token;
```

```
/* ** prototypes ** */
token_t getNextToken(void);
int term(void);
int factor(void);
int expr(void);

int main()
{
  int value;
  while ((token = getNextToken()) != EOF) {
    if (token == '\n')
      continue;
    value = expr();
    printf("%d\n", value);
  }
  exit(0);
}

/** expr() parses and evaluates an arithmetic expression, returning its
 * numerical value.  Arithmetic expressions are defined with the BNF:
 *
 *  <expr> ::= <term> { <addop> <term> }
 *
 * ENTRY CONDITIONS: The next unprocessed token must be already
 *                   available in the token global variable.
 * EXIT CONDITIONS:  The next unprocessed token will be placed
 *                   in the token global variable.
 * @return The value of the expression.
 */
int expr(void)
{
  int value,  valueRight;
  token_t opToken;

  value = term();
  while((opToken = token) == '+' || opToken == '-') {
    token = getNextToken();
    valueRight = term();
```

```
    if(opToken == '+')
      value = value + valueRight;
    else
      value = value - valueRight;
  }
  return value;
}

/** getNextToken() reads the next character from stdin (ignoring whitespace)
 * and returns it.
 *
 * @return the next character that is not a space or a tab.
 */

token_t getNextToken(void)
{
  int ch;
  /* Skip spaces and tabs */
  while (((ch = getchar()) == ' ') || (ch == '\t'))
    ;
  return ch;
}

/** term() parses and evaluates a <term> as defined by the BNF:
 *  <term> ::= <factor> { <mulop> <factor> }
 *
 * ENTRY CONDITIONS: The next unprocessed token must be already
 *                   available in the token global variable.
 * EXIT CONDITIONS:  The next unprocessed token will be placed
 *                   in the token global variable.
 *
 * @return The value of the term.
 */
int term(void)
{
  int value, valueRight;
  token_t opToken;
```

```
  value = factor();
  while((opToken = token) == '*' || token == '/') {
    token = getNextToken();
    valueRight = factor();
    if(opToken == '*')
      value = value * valueRight;
    else
      value = value / valueRight;
  }
  return value;
}

/** factor() parses and evaluates a <factor> as defined by the BNF:
 *  <factor> ::= <num> | '(' <expr> ')'
 *
 * ENTRY CONDITIONS: The next unprocessed token must be already
 *                   available in the token global variable.
 * EXIT CONDITIONS:  The next unprocessed token will be placed
 *                   in the token global variable.
 *
 * @return The value of the factor.
 */

int factor(void)
{
  int value;

  if (isdigit(token)) {
    value = token - '0';
    token = getNextToken();
    return value;
  } else {
    assert(token == '(');
    token = getNextToken();
    value = expr();
    assert(token == ')');
    token = getNextToken();
  }
```

```
  return value;
}
```

E.3.4 nounPhraseWordCounter.c

```
/* Copyright (C) 1999 Ken Clowes (kclowes@ee.ryerson.ca) */

/** This program parses noun phrases writing
 * "Good: 'n' words" for those that are grammatically
 * correct or "Bad" otherwise.  The noun phrases
 * (zero or more) are read from stdin, one phrase per line.
 * Empty lines are ignored.
 *
 * The noun phrases are described by the following BNF:
 * <pre>
 * <noun_phrase> ::= <article> { <adjective> } <noun>
 * <article> ::= 'the'
 *             | 'a'
 * <adjective> ::= 'hungry'
 *               | 'black'
 *               | 'white'
 *               | 'cute'
 * <noun> ::= 'cat' | 'dog'
 * </pre>
 *
 * The exit code is zero unless any of the words are too long.
 *
 * If this file is compiled with either "TEST_TOKENIZER" or
 * "TEST_TOKENIZER_TWO" defined, a different executable is created.
 * To find out more about these: UTSL (i.e. "Use The Source, Luke")...
 */

#include <stdio.h>
#include <ctype.h>

#define MAX_WORD_LEN 100 /* Any word longer than this will cause an
                            immediate exit with a "1" exit code */
```

```
#ifndef TEST_TOKENIZER
#define DEFAULT
#endif

/* ** typedefs ** */
typedef char * token_t;

/* ** Globals ** */
token_t token;

/** getNextToken() reads the next word from stdin and returns a pointer
 * to it.  Whitespace (spaces and tabs) are ignored.  If the end of file
 * is encountered, an empty string is returned.  The end-of-line
 * indicator is returned as the string "\n".
 *
 * @return a pointer to the string or "" or "\n" for EOF and newline.
 */

token_t getNextToken(void)
{
  static char t[MAX_WORD_LEN + 1];
  int ch, i;
  /* Skip white space */
  while((((ch = getchar()) != '\n') && isspace(ch)))
    ;
  switch (ch) {
  case EOF:
    t[0] = '\0';
    break;
  case '\n':
    t[0] = ch;
    t[1] = '\0';
    break;
  default:
    t[0] = ch;
    for(i = 1; ((ch = getchar()) != '\n') && !isspace(ch); i++) {
      if (i >= MAX_WORD_LEN) {
        fprintf(stderr, "Aborting: cannot tokenize words longer "
```

```
                "than %d characters\n", MAX_WORD_LEN);
        exit(1);
      }
      t[i] = ch;
    }
    if (ch == '\n')
      ungetc(ch, stdin);
    t[i] = '\0';
    break;
  }
  return t;
}

/** article() parses an article as defined by:
 * <pre>
 *<article> ::= 'the' | 'a'
 * </pre>
 * ENTRY CONDITIONS: The next unprocessed token must be already
 *                   available in the token global variable.
 * EXIT CONDITIONS:  The next unprocessed token will be placed
 *                   in the token global variable.
 * @return 1 if the token is an article; otherwise 0.
 */

int article(void)
{
  if ((strcmp(token, "a") == 0) || (strcmp(token, "the") == 0)) {
    token = getNextToken();
    return 1;
  }
  return 0;
}

/* isAdjective returns 1 if "word" is a valid adjective; 0 if not. */
int isAdjective(char * word)
{
  static char * adjs[] = {"white", "black", "hungry", "cute"};
  int found = 0;
```

```
  int i;
  for(i = 0; i < sizeof(adjs) / sizeof(char *); i++) {
    if (strcmp(word, adjs[i]) == 0) {
      found = 1;
      break;
    }
  }
  return found;
}

/** adjective() parses an adjective as defined by:
 * <pre>
 *<adjective> ::= 'white' | 'black' | 'hungry' | 'cute'
 * </pre>
 * ENTRY CONDITIONS: The next unprocessed token must be already
 *                   available in the token global variable.
 * EXIT CONDITIONS:  The next unprocessed token will be placed
 *                   in the token global variable.
 * @return the number of adjectives parsed.
 */
int adjective(void)
{
  int r;
  r = isAdjective(token);
  if (r == 1)
    token = getNextToken();
  return r;
}

/** noun() parses an noun as defined by:
 * <pre>
 *<noun> ::= 'cat' | 'dog'
 * </pre>
 * ENTRY CONDITIONS: The next unprocessed token must be already
 *                   available in the token global variable.
 * EXIT CONDITIONS:  The next unprocessed token will be placed
 *                   in the token global variable.
 * @return 1 if the token is an noun; otherwise 0.
```

```
 */
int noun(void)
{
  if ((strcmp(token, "dog") == 0) || (strcmp(token, "cat") == 0)) {
    token = getNextToken();
    return 1;
  }
  return 0;
}

/** noun_phrase() parses a noun phrase defined by the BNF:
 * <pre>
 * <noun_phrase> ::= <article> { <adjective> } <noun>
 * </pre>
 *
 * ENTRY CONDITIONS: The next unprocessed token must be
 *                   already available in the token
 *                   global variable.
 * EXIT CONDITIONS:  The next unprocessed token will be placed
 *                   in the token global variable.
 * @return The number of words in the noun phrase
 *         (must be at least 2) or a negative number
 *         if a parse error is detected.
 */
int noun_phrase(void)
{
  int nWords;

  nWords = article();
  if(nWords == 0) {
    return -1;
  }
  while(adjective())
    nWords++;
  if(!noun()) {
    return -2;
  }
  return nWords+1;
```

```
}

#ifdef DEFAULT
int main()
{
  int nWords;
  while(strcmp(token = getNextToken(), "")) { /* while NOT end-of-file */
    if (!strcmp(token, "\n")) /* Ignore empty lines */
      continue;
    if((nWords = noun_phrase()) > 1)
      printf("Good: %d words\n", nWords);
    else {
      printf("Bad\n");
      /* Skip rest of line */
      while(strcmp(token, "\n") != 0)
        token = getNextToken();
    }
  }
  exit(0);
}
#endif

#ifdef TEST_TOKENIZER
int main()
{
  while(strcmp((token = getNextToken()), "")) {
    printf("Next token: %s\n", token);
  }
  exit(0);
}
#endif
```

E.4 ADTs

Source code for Chapter 5. This source code can also be found in the directory `src/ADT`.

E.4.1 README

```
This directory contains the source code for programs
discussed in Chapter 5---Abstract Data Types---of the book
"Engineering Algorithms and Data Structures".

README    --- this file
Makefile  --- the Makefile (what else)

IntLLBag.c  --- A linked list (LL) implementation
                of a Bag containing ints.
IntLLBag.h  --- User header file for the implementation.

IntVBag.c --- A resizeabe array (vector, V) implementation
                of a Bag containing ints.
IntVBag.h  --- User header file for the implementation.

IntBag.c --- A general implementation of a Bag of ints.
IntBag.h  --- User header file for the implementation.
IntBagP.h --- The private header file for the implementation.

IntLLBag2.c --- An ADT implementation of IntLLBag

simpleTestIntLLBag.c --- A very simple test of IntLLBag.
simpleTIntBag.c --- A very simple test of IntBag,
                    IntLLBag2 and IntVBag2.

test1IntLLBag.c
testIntBag.c
testIntLLBag.c
testIntVBag.c
```

E.4.2 Makefile

```
CFLAGS=-Wall -g -ansi -pedantic -I../../include
EXECS=testIntLLBag test1IntLLBag testIntVBag testIntBag \
      simpleTestIntLLBag simpleTIntBag
CC=gcc

all: ${EXECS}

testIntLLBag: testIntLLBag.o IntLLBag.o eprintf.o
gcc -o testIntLLBag testIntLLBag.o IntLLBag.o eprintf.o

test1IntLLBag: test1IntLLBag.o IntLLBag.o eprintf.o
gcc -o test1IntLLBag test1IntLLBag.o IntLLBag.o eprintf.o

testIntVBag: testIntVBag.o IntVBag.o eprintf.o
gcc -o testIntVBag testIntVBag.o IntVBag.o  eprintf.o

testIntBag: testIntBag.o IntBag.o IntLLBag2.o eprintf.o
gcc -o testIntBag testIntBag.o IntBag.o IntLLBag2.o eprintf.o

simpleTestIntLLBag: simpleTestIntLLBag.o IntLLBag.o eprintf.o
gcc -o simpleTestIntLLBag simpleTestIntLLBag.o IntLLBag.o eprintf.o

simpleTIntBag: simpleTIntBag.o IntBag.o IntLLBag2.o eprintf.o
gcc -o simpleTIntBag simpleTIntBag.o IntLLBag2.o IntBag.o eprintf.o

clean:
-@rm -f *~ *.dvi *.log *.ps *.log *.aux *.o ${EXECS} junk*
```

E.4.3 IntLLBag.h

```
#ifndef _IntLLBag_H
#define _IntLLBag_H
typedef void * IntLLBag;
```

```
/** Create a new IntLLBag (i.e. a "bag" of ints implemented as a
 * Linked List.)
 *
 * @return - A new IntLLBag or NULL if one cannot be created.
 */

IntLLBag newIntLLBag(void);

/** Add an integer to an IntLLBag.
 *
 *  @param b The Bag that will be added to.
 *  @param i The integer to add.
 */

void addIntLLBag(IntLLBag b, int i);

/** Remove an integer from an IntLLBag. If the Bag is empty,
 *  the program exits with a non-zero status.
 *
 *  @param b The Bag that an int will be removed from.
 *  @return  The removed integer.
 */

int removeIntLLBag(IntLLBag b);

/** Determine the number of elements in an IntLLBag.
 *
 *  @param b The Bag whose size is determined.
 *  @return  The number of elements in the Bag.
 */
unsigned int getSizeIntLLBag(IntLLBag b);

/** Destroy a previously created IntLLBag, releasing all its
 *  resources.
 *
 *  @param b The Bag to delete.
 */
void destoryIntLLBag(IntLLBag b);
```

```
#endif /* #ifndef _IntLLBag_H */
```

E.4.4 IntLLBag.c

```
#include <stdio.h>
#include <stdlib.h>
#include "IntLLBag.h"
#include "eprintf.h"

typedef struct _LList _LList, *_LListPtr;

struct _LList {
  int data;
  _LListPtr next;
};

typedef struct _IntLLBag _IntLLBag, *_IntLLBagPtr;
struct _IntLLBag {
  _LListPtr head;
  int size;
};

/** Create a new IntLLBag (i.e. a "bag" of ints implemented as a
 * Linked List.)
 *
 * @return - A new IntLLBag or NULL if one cannot be created.
 */
IntLLBag newIntLLBag(void)
{
  _IntLLBagPtr b;
  b = malloc(sizeof(_IntLLBag));
  if (b == NULL)
    return NULL;
  b->head = ( _LListPtr )  NULL;
  b->size = 0;
  return (IntLLBag) b;
}
```

```
/** Add an integer to an IntLLBag.
 *
 *  @param b The Bag that will be added to.
 *  @param i The integer to add.
 */
void addIntLLBag(IntLLBag b, int i)
{
  _LListPtr item;
  _IntLLBagPtr _b = (_IntLLBagPtr) b;

  item = malloc(sizeof(_LList));
  item->data = i;
  item->next = _b->head;
  _b->size++;
  _b->head = item;
  return;
}

/** Remove an integer from an IntLLBag. If the Bag is empty,
 *  the program exits with a non-zero status.
 *
 *  @param b The Bag that an int will be removed from.
 *  @return  The removed integer.
 */
int removeIntLLBag(IntLLBag b)
{
  int r;
  _LListPtr item;
  _IntLLBagPtr _b = (_IntLLBagPtr) b;

  if (_b->size <= 0) {
    eprintf("Fatal error, removing from empty bag\n");
  }
  _b->size--;

  item = _b->head;
  r = item->data;
```

```
  _b->head = item->next;
  free(item);
  return r;
}

/** Determine the number of elements in an IntLLBag.
 *
 *  @param b The Bag whose size is determined.
 *  @return  The number of elements in the Bag.
 */
unsigned int getSizeIntLLBag(IntLLBag b)
{
  return ((_IntLLBagPtr)b)->size;
}

/** Destroy a previously created IntLLBag, releasing all its
 *  resources.
 *
 *  @param b The Bag to delete.
 */
void destoryIntLLBag(IntLLBag b)
{
  _IntLLBagPtr _b = (_IntLLBagPtr) b;

  while(_b->size > 0) {
    removeIntLLBag(b);
  }
  free(_b);
  return;
}
```

E.4.5 IntVBag.h

```
#ifndef _IntVBag_H
#define _IntVBag_H
typedef void * IntVBag;

/** Create a new IntVBag (i.e. a "bag" of ints implemented as a
```

```
 * Linked List.)
 *
 * @return - A new IntVBag or NULL if one cannot be created.
 */

IntVBag newIntVBag(void);

/** Add an integer to an IntVBag.
 *
 *  @param b The Bag that will be added to.
 *  @param i The integer to add.
 */

void addIntVBag(IntVBag b, int i);

/** Remove an integer from an IntVBag. If the Bag is empty,
 *  the program exits with a non-zero status.
 *
 *  @param b The Bag that an int will be removed from.
 *  @return  The removed integer.
 */

int removeIntVBag(IntVBag b);

/** Determine the number of elements in an IntVBag.
 *
 *  @param b The Bag whose size is determined.
 *  @return  The number of elements in the Bag.
 */
unsigned int getSizeIntVBag(IntVBag b);

/** Destroy a previously created IntVBag, releasing all its
 *  resources.
 *
 *  @param b The Bag to delete.
 */
void destoryIntVBag(IntVBag b);
```

```
#endif /* #ifndef _IntVBag_H */
```

E.4.6 IntVBag.c

```
#include <stdlib.h>
#include "IntVBag.h"
#include "eprintf.h"

typedef struct _IntVBag _IntVBag,* _IntVBagPtr;
struct _IntVBag {
  int * data;
  int size;
  int maxSize;
};

/** Create a new IntVBag (i.e. a "bag" of ints implemented as a
 * Linked List.)
 *
 * @return - A new IntVBag or NULL if one cannot be created.
 */

IntVBag newIntVBag(void)
{
  _IntVBagPtr b;

  b = malloc(sizeof(_IntVBag));
  if (b == NULL)
    return NULL;
  b->size = 0;
  b->maxSize = 0;
  b->data = NULL;
  return (IntVBag) b;
}

/** Add an integer to an IntVBag.
 *
 *  @param b The Bag that will be added to.
 *  @param i The integer to add.
```

```
 */

void addIntVBag(IntVBag b, int i)
{
  _IntVBagPtr _b = (_IntVBagPtr) b;

  if (_b->size >= _b->maxSize) {
    _b->maxSize = _b->maxSize==0 ? 1 : 2*_b->maxSize;
    _b->data = erealloc(_b->data, _b->maxSize*sizeof(int));
  }
  _b->data[_b->size] = i;
  _b->size++;
  return;
}

/** Remove an integer from an IntVBag. If the Bag is empty,
 *  the program exits with a non-zero status.
 *
 *  @param b The Bag that an int will be removed from.
 *  @return  The removed integer.
 */
int removeIntVBag(IntVBag b)
{
  _IntVBagPtr _b = (_IntVBagPtr) b;

  if (_b->size <= 0) {
    eprintf("Fatal error, removing from empty bag\n");
  }
  return _b->data[--_b->size];
}

/** Determine the number of elements in an IntVBag.
 *
 *  @param b The Bag whose size is determined.
 *  @return  The number of elements in the Bag.
 */
unsigned int getSizeIntVBag(IntVBag b)
{
```

```
  _IntVBagPtr _b = (_IntVBagPtr) b;
  return _b->size;
}

/** Destroy a previously created IntVBag, releasing all its
 *  resources.
 *
 *  @param b The Bag to delete.
 */
void destoryIntVBag(IntVBag b)
{
  _IntVBagPtr _b = (_IntVBagPtr) b;
  free(_b->data);
  _b->size = 0;
  _b->data = NULL;
  free(_b);
  return;
}
```

E.4.7 IntBag.h

```
#ifndef _IntBag_H
#define _IntBag_H
typedef void * IntBag;

/** Create a new IntBag
 *   (i.e. a "bag" of ints--default implementation)
 *
 * @return - A new IntBag or NULL if one cannot be created.
 */

IntBag newIntBag(void);

/** Create a new IntLLBag (i.e. a "bag" of ints implemented as a
 * Linked List.)
 *
 * @return - A new IntLLBag or NULL if one cannot be created.
 */
```

```
IntBag newIntLLBag(void);

/** Add an integer to an IntBag.
 *
 *  @param b The Bag that will be added to.
 *  @param i The integer to add.
 */

void addIntBag(IntBag b, int i);

/** Remove an integer from an IntBag. If the Bag is empty,
 *  the program exits with a non-zero status.
 *
 *  @param b The Bag that an int will be removed from.
 *  @return  The removed integer.
 */

int removeIntBag(IntBag b);

/** Determine the number of elements in an IntBag.
 *
 *  @param b The Bag whose size is determined.
 *  @return  The number of elements in the Bag.
 */
unsigned int getSizeIntBag(IntBag b);

/** Destroy a previously created IntBag, releasing all its
 *  resources.
 *
 *  @param b The Bag to delete.
 */
void destoryIntBag(IntBag b);

#endif /* #ifndef _IntBag_H */
```

E.4.8 IntBagP.h

```
#include "IntBag.h"
#ifndef _IntBagP_H
#define _IntBagP_H 1

#define private static
#define public

enum {
  addMethod = 0,
  getSizeMethod,
  removeMethod,
  destroyMethod
};

typedef struct _IntBag _IntBag, * _IntBagPtr;

typedef void * (*_method)(IntBag b, ...);

struct _IntBag {
  void * data;
  union {
    void * ptr;
    int intVal;
  } state;
  _method *methods;
};

#endif  /* #ifdef _IntBagP_H */
```

E.4.9 IntBag.c

```
#include <stdio.h>
#include <stdlib.h>
#include "IntBagP.h"
#include "eprintf.h"
```

```
/** Create a new IntBag
 *   (i.e. a "bag" of ints--default implementation)
 *
 * @return - A new IntBag or NULL if one cannot be created.
 */
public IntBag newIntBag(void)
 {
  return (IntBag) newIntLLBag();
}

/** Add an integer to an IntBag.
 *
 *  @param b The Bag that will be added to.
 *  @param i The integer to add.
 */
void addIntBag(IntBag b, int i)
{
  _IntBagPtr _b;

  _b = (_IntBagPtr) b;
  _b->methods[addMethod](b, i);
  return;
}

/** Remove an integer from an IntBag. If the Bag is empty,
 *  the program exits with a non-zero status.
 *
 *  @param b The Bag that an int will be removed from.
 *  @return  The removed integer.
 */
int removeIntBag(IntBag b)
{
  _IntBagPtr _b;

  _b = (_IntBagPtr) b;
  return (int) _b->methods[removeMethod](b);
}
```

```
/** Determine the number of elements in an IntBag.
 *
 *  @param b The Bag whose size is determined.
 *  @return  The number of elements in the Bag.
 */
unsigned int getSizeIntBag(IntBag b)
{
  _IntBagPtr _b;

  _b = (_IntBagPtr) b;
  return (unsigned int) _b->methods[getSizeMethod](b);
}

/** Destroy a previously created IntBag, releasing all its
 *  resources.
 *
 *  @param b The Bag to delete.
 */
void destoryIntBag(IntBag b)
{
  _IntBagPtr _b;

  _b = (_IntBagPtr) b;
  (void) _b->methods[destroyMethod](b);
  return;
}
```

E.4.10 IntLLBag2.c

```
#include <stdio.h>
#include <stdlib.h>
#include "IntBagP.h"
#include "eprintf.h"

/* The _IntBag data structure is interpreted as follows:
 * struct _IntBag {
```

```
 *   void * data;       --- the "head" of the list of ints
 *   union {
 *     void * ptr;
 *     int intVal;      --- the "size" of the list
 *   } state;
 *   _method *methods;  --- the array of functions
 *                          operating on the list
 * };
*/

typedef struct _LList _LList, *_LListPtr;
struct _LList {
  int data;
   _LListPtr next;
};

public IntBag newIntLLBag(void);

private void _addIntLLBag(IntBag b, int i);
private int _removeIntLLBag(IntBag b);
private int _getSizeIntLLBag(IntBag b);

private _method theseMethods[] = {
  (_method) &_addIntLLBag,
  (_method) &_getSizeIntLLBag,
  (_method) &_removeIntLLBag
};

private void _addIntLLBag(IntBag b, int i)
{
  _LListPtr item;
  _IntBagPtr _b = (_IntBagPtr) b;

  item = malloc(sizeof(_LList));
  item->data = i;
  item->next = _b->data;
   _b->state.intVal++;    /* _b->state.intVal IS size */
  _b->data = item;        /*_b->data IS head of list */
```

```
  return;
}

/** Remove an integer from an IntLLBag. If the Bag is empty,
 *  the program exits with a non-zero status.
 *
 *  @param b The Bag that an int will be removed from.
 *  @return  The removed integer.
 */
private int _removeIntLLBag(IntBag b)
{
  int r;
  _LListPtr item;
  _IntBagPtr _b = (_IntBagPtr) b;

  if (_b->state.intVal <= 0) { /* _b->state.intVal IS size */
    eprintf("Fatal error, removing from empty bag\n");
  }
  _b->state.intVal--; /* b->state.intVal IS size */

  item = _b->data;    /* _b->data IS head of list */
  r = item->data;
  _b->data = item->next;  /* _b->data IS head of list */
  free(item);
  return r;
}

private int _getSizeIntLLBag(IntBag b)
{
  _IntBagPtr _b = (_IntBagPtr) b;

  return _b->state.intVal;  /* _b->state.intVal IS size */
}

/** Create a new IntBag using a Linked List implementation
 *
 * @return - A new IntBag or NULL if one cannot be created.
 */
```

```
public IntBag newIntLLBag(void)
{
  _IntBagPtr b;
  b = malloc(sizeof(_IntBag));
  if (b == NULL)
    return NULL;
  b->data =  NULL;    /* b->data IS head of list */
  b->state.intVal = 0;      /* b->state.intVal IS size */
  b->methods = theseMethods;
  return (IntBag) b;
}
```

E.4.11 IntVBag2.c

```
#include <stdio.h>
#include <stdlib.h>
#include "IntBagP.h"
#include "eprintf.h"

typedef struct _IntVBag _IntVBag,* _IntVBagPtr;
struct _IntVBag {
  int * data;
  int size;
  int maxSize;
};

public IntBag newIntVBag(void);

private void _addIntVBag(IntBag b, int i);
private int _removeIntVBag(IntBag b);
private int _getSizeIntVBag(IntBag b);
```

```
private _method theseMethods[] = {
  (_method) &_addIntVBag,
  (_method) &_getSizeIntVBag,
  (_method) &_removeIntVBag
};

private void _addIntVBag(IntBag b, int i)
{

  return;
}

/** Remove an integer from an IntVBag. If the Bag is empty,
 *  the program exits with a non-zero status.
 *
 *  @param b The Bag that an int will be removed from.
 *  @return  The removed integer.
 */
private int _removeIntVBag(IntBag b)
{
  int r;
  _VistPtr item;
  _IntBagPtr _b = (_IntBagPtr) b;

  if (_b->_state._intVal <= 0) {
    eprintf("Fatal error, removing from empty bag\n");
  }
  _b->_state._intVal--;

  item = _b->_data;
  r = item->data;
  _b->_data = item->next;
  free(item);
  return r;
}

private int _getSizeIntVBag(IntBag b)
{
```

```
  _IntBagPtr _b = (_IntBagPtr) b;

  return _b->_state._intVal;
}

/** Create a new IntVBag (i.e. a "bag" of ints implemented as a
 * Linked List.)
 *
 * @return - A new IntVBag or NULL if one cannot be created.
 */
public IntBag newIntVBag(void)
{
  _IntBagPtr b;
  b = malloc(sizeof(_IntBag));
  if (b == NULL)
    return NULL;
  b->_data =  NULL;
  b->_state._intVal = 0;
  b->_methods = theseMethods;
  return b;
}
```

E.4.12 simpleTestIntLLBag.c

```
#include <stdio.h>
#include "IntLLBag.h"

int main(int argc, char * argv[])
{
  IntLLBag b;

  b = newIntLLBag();
```

```
  addIntLLBag(b, 3);
  addIntLLBag(b, 1);
  addIntLLBag(b, 4);
  addIntLLBag(b, 1);

  while(getSizeIntLLBag(b))
    printf("Removed: %d\n", removeIntLLBag(b));
  exit(0);
}
```

E.4.13 simpleTIntBag.c

```
#include <stdio.h>
#include "IntBag.h"

int main(int argc, char * argv[])
{
  IntBag b1, b2;

  b1 = newIntBag(); b2 = newIntLLBag();

  addIntBag(b1, 3);  addIntBag(b2, 3);
  addIntBag(b1, 1);  addIntBag(b2, 1);
  addIntBag(b1, 4);  addIntBag(b2, 4);
  addIntBag(b1, 1);  addIntBag(b2, 1);

  while(getSizeIntBag(b1))
    printf("Removed: %d (from b1) and %d (from b2)\n",
   removeIntBag(b1), removeIntBag(b2));
  exit(0);
}
```

E.5 Trees

Source code for Chapter 7. This source code can also be found in the directory `src/trees`.

E.5.1 README

```
This directory contains the files associated with
Chapter 7---Trees---in the book "Engineering
Algorithms and Data Structures".

README -- this file
Makefile -- the makefile (what else)
myFamily.c -- Initilaizes data structs for "my family"
traverse.c -- pre- and post-order traversal
trees.h    -- header file for tree data structures
```

E.5.2 Makefile

```
CFLAGS=-Wall -g
EXECS= myFamily
CC=gcc

all: ${EXECS}

myFamily: myFamily.o traverse.o
gcc -o myFamily myFamily.o traverse.o

clean:
-@rm -f *~ *.dvi *.log *.ps *.log *.aux *.o ${EXECS} *.exe
```

E.5.3 trees.h

```
typedef struct TreeNode TreeNode, * TreeNodePtr;
typedef char * NodeInfo;  /* For example */

#define MAX_KIDS 10

struct TreeNode {
  NodeInfo info;
  int nKids;
  TreeNodePtr  kids[MAX_KIDS];
```

```
};

extern void postOrder(TreeNodePtr t, int depth);
extern void preOrder(TreeNodePtr t, int depth);
```

E.5.4 myFamily.c

```
#include <stdio.h>
#include "trees.h"

TreeNode me, mom, dad, momsMom, momsDad, dadsDad, dadsMom;

TreeNode me = {"ME", 2, {&mom, &dad}};

TreeNode mom = {"Mom", 2, {&momsMom, &momsDad}};
TreeNode dad = {"Dad", 2, {&dadsMom, &dadsDad}};
TreeNode momsMom = {"Mom's mom", 2, {NULL, NULL}};
TreeNode momsDad = {"Mom's dad", 2, {NULL, NULL}};
TreeNode dadsMom = {"Dad's mom", 2, {NULL, NULL}};
TreeNode dadsDad = {"Dad's dad", 2, {NULL, NULL}};


int main()
{
  printf("Post Order:\n");
  postOrder(&me, 0);
  printf("\nPre Order:\n");
  preOrder(&me, 0);
  exit(0);
}
```

E.5.5 traverse.c

```
#include <stdio.h>
#include "trees.h"

void postOrder(TreeNodePtr t, int depth)
```

```
{
  int i, k;
  if(t != NULL) {
    for(k = 0; k < t->nKids; k++)
      postOrder(t->kids[k], depth+1);
    for(i = 0; i < depth; i++)
      printf("    ");
    printf("%s\n", t->info);
  }
}

void preOrder(TreeNodePtr t, int depth)
{
  int i, k;

  if(t != NULL) {
    for(i = 0; i < depth; i++)
      printf("    ");
    printf("%s\n", t->info);
    for(k = 0; k < t->nKids; k++)
      preOrder(t->kids[k], depth+1);
  }
}
```

E.6 Source code listings for Digital Simulator

E.6.1 A sample `main` function

The following C source code shows you how to write a main driver function for a specific circuit. You can use this as a template fro writing your own main function. The only things that should change are the declarations of the wires and blocks, their instantiation, and any specific events you want to add to the queue before performing the simulation.

The circuit implemented in this example is the simple clock generator circuit in Figure 14.3. This file is available as `mainClock.c`.

```
/* Copyright (C) 1999 Ken Clowes (kclowes@ee.ryerson.ca) */
```

```
/**REVISION HISTORY:
  * Created: March 13, 1999 (kclowes@ee.ryerson.ca)
  */
#include <stdio.h>
#include <string.h>

#include "wire.h"
#include "eventQ.h"
#include "nand.h"

/* *** Global variables *** */
char * programName;  /* The command used to invoke the program */
int verbosity = 0;   /* Additional info to <stderr> if non-zero */
EventQ_t eventQ;     /* The event queue used by the simulator */


void usage(void)
{
  fprintf(stderr, "Usage: %s [-v]\n", programName);
  exit(1);
}

/**
  * The main driver for the simulator. It sets up the circuit and invokes
  * the main simulation loop.
  * The only command line option that is recognized is "-v".  If this
  * option is present, the global variable "verbosity" is set to 1.
  * Other modules may use this value to print additional information to
  * <stderr>.
  *
  * The circuit set up here is a simple clock generator using an
  * inverter.
  *
  * @param argc--the number of command line arguments
  * @param argv--the actual arguments
  * @return Normally exits with an error code of 0 unless incorrectly
  *         invoked or a run time error occurs.
```

```
  */
int main(int argc, char * argv[])
{
  unsigned long nEvents;
  /* Declare the wires and blocks used */
  Wire_t clock;
  Block_t inverter;

  programName = argv[0];

  if(argc > 2) {
    usage();
  }
  if (argc == 2) {
    if (strcmp(argv[1], "-v") != 0) {
      usage();
    }
    verbosity = 1;
    fprintf(stderr, "VERBOSE output to <stderr>!\n");
  }
  eventQ = newEventQ();

  /* Create the circuit */
  clock = newWire("Clock", 0);
  inverter = newNand("inv", 50, clock, clock, NULL);

  /* Add any additional events to the queue */

  /* Simulate the circuit */
  nEvents = simulate(1000);

  /* Print the statistics to stderr */
  fprintf(stderr, "Simulation terminated after %lu events.\n"
          "The simulation time was %ld\n"
          "There were %d events still in the queue\n",
          nEvents, eventQ->currentTime, eventQSize(eventQ));
  exit(0);
}
```

E.6.2 The simulation algorithm (simulate.c)

The basic simulation algorithm is shown below. It is circuit-independent, so you should never have to change it.

```
/* Copyright (C) 1999 Ken Clowes (kclowes@ee.ryerson.ca) */

/**REVISION HISTORY:
  * Created: March 13, 1999 (kclowes@ee.ryerson.ca)
  */
#include <stdlib.h>
#include <stdio.h>
#include <assert.h>
#include "simulate.h"
#include "eventQ.h"

/**
  * "simulate" impelements the main simulation loop.
  * It prints to <stdout> each event that changes a wire's value
  * in the following format:
  *    <wire_name> <wire_value> "at time" <current_simulation_time>
  *
  * @param maxTime The maximum simulated time allowable for the simulation
  * @return The number of events handled.
  */
unsigned long simulate(long maxTime)
{
  Event_t event;
  unsigned long nEvents = 0;
  Wire_t wire;
  Value_t value;
  Block_t block;

  assert(maxTime >= 0);
  if (verbosity) {
    fprintf(stderr, "\"simulate(%ld)\" invoked: ", maxTime);
```

```
    fprintf(stderr, "simTime: %ld, ", eventQ->currentTime);
    fprintf(stderr, "eventQ: %p\n", eventQ);
    fflush(stderr);
  }
  while((eventQ->currentTime < maxTime) &&
        (event = removeNextEvent(eventQ)) != NULL) {
    nEvents++;
    if (verbosity) {
      fprintf(stderr, "Got event %lu: time (%ld), wire (%s), value (%d)\n",
              nEvents, getEventTime(event), getWireName(getEventWire(event)),
              getEventValue(event));
    }
    wire = getEventWire(event);
    value = getEventValue(event);
    if(getWireValue(wire) == value) {
      if (verbosity) {
        fprintf(stderr, "Ignoring event (time=%ld, wire=%s, value=%d)\n",
                getEventTime(event), getWireName(wire), value);
        free(event);
      }
      continue; /* Ignore events that change nothing. */
    }
    free(event);
    setWireValue(wire, value);
    /* #define NO_OUTPUT     */
#ifndef NO_OUTPUT
    printf("%s %d at time %ld\n", getWireName(wire), getWireValue(wire),
           eventQ->currentTime);
#endif
    resetWireDependents(wire);
    while((block = getNextWireDependent(wire)) != NULL) {
      evaluateBlock(block);
    }
  }
  return nEvents;
}
```

E.6.3 value.h

```
/* Copyright (C) 1999 Ken Clowes (kclowes@ee.ryerson.ca) */

/**REVISION HISTORY:
  * Created: March 13, 1999 (kclowes@ee.ryerson.ca)
  */

#ifndef VALUE_H
#define VALUE_H
#include "simulate.h"

#define MAX_VALUE (1)
#define MIN_VALUE (0)

extern int isValidValue(Value_t v);

#endif /* VALUE_H */
```

E.6.4 wire.h

```
/* Copyright (C) 1999 Ken Clowes (kclowes@ee.ryerson.ca) */

/**REVISION HISTORY:
  * Created: March 13, 1999 (kclowes@ee.ryerson.ca)
  */

#ifndef WIRE_H
#define WIRE_H
#include "simulate.h"
#include "value.h"
#include "block.h"
#include "eventQ.h"

/* GENERAL BUGS:
 * All of the following functions may fail catastrophically at run
 time if the Wire_t parameter is NOT a real wire created with the
 "newWire" interface. */
```

```
/** "newWire" creates a new wire with the specified <name> and initial
  * value.
  *
  * The "name" may be a NULL pointer, an empty string or a
  * user-defined name (which may not, however, begin with the
  * characters "0x").
  *
  * If <name> is is "" or NULL, the actual name of the wire will
  * be given a name beginning with "0x" that is guaranteed to be
  * unique.
  *
  *  If <name> is defined, the wire will be given the defined name;
  * there is NO GUARANTEE that this name is unique.
  *
  * @param name --- the name of the wire (possibly "" or NULL)
  * @param initialValue --- the initial value of the wire
  * @return A reference to the wire.
  */
extern Wire_t newWire(char * name, Value_t initialValue);

/** "addWireDependent" adds a BLOCK that has "this" wire as an
  * input.
  * @param w --- the WIRE whose dependents are being added to.
  * @param b --- the BLOCK that is dependent on this wire.
  */

extern void addWireDependent(Wire_t w, Block_t b);

/** "getWireValue" returns the curent value of a wire.
  * @param w ---  the WIRE to examine.
  * @return --- The current value of the wire.
  */

extern Value_t getWireValue(Wire_t w);

/** "setWireValue" sets the current value of the wire to the specified
  * value.
```

```
  * @param w --- the WIRE to modify.
  * @param v --- the new (current) value of the wire.
  */

extern void setWireValue(Wire_t w, Value_t v);

/* The following functions (methods) relate to the collection of
 * BLOCKs that are dependent on a WIRE.
 */

/** "resetWireDependents" re-initializes the list of dependent blocks
  * so that subsequent calls to "getNextWireDependent" will return
  * all of the dependents (and then NULL).
  *
  * @param w --- the WIRE whose dependent BLOCKs are being reset
  *
  *
  */
extern void resetWireDependents(Wire_t w);

/** "getNextWireDependent" returns the next BLOCK that depends on the
  *    WIRE.  If there are no more dependent blocks, it returns NULL.
  */
extern Block_t getNextWireDependent(Wire_t w);
extern char * getWireName(Wire_t w);
#endif /* WIRE_H */
```

E.6.5 block.h

```
/* Copyright (C) 1999 Ken Clowes (kclowes@ee.ryerson.ca) */

/**REVISION HISTORY:
  * Created: March 13, 1999 (kclowes@ee.ryerson.ca)
  */

#ifndef BLOCK_H
#define BLOCK_H
#include "simulate.h"
```

```
#include "wire.h"

/** "newBlock" creates and returns a new block with the specified name.
  * The rules for specifying a name are the same as those for wires.
  */
extern Block_t newBlock(char * name);
/** "evaluateBlock" causes the block to re-examine its inputs and generate
  * any events on its outputs that are appropriate.
  */
extern void evaluateBlock(Block_t b);

/** "addInputToBlock" adds the specified wire as an input to the block.
  */
extern void addInputToBlock(Block_t b, Wire_t w);
/** "addOutputToBlock" adds the specified wire as an output to the block.
  */
extern void addOutputToBlock(Block_t b, Wire_t w);
#endif /* BLOCK_H */
```

E.6.6 nand.h

```
/* Copyright (C) 1999 Ken Clowes (kclowes@ee.ryerson.ca) */

/**REVISION HISTORY:
  * Created: March 13, 1999 (kclowes@ee.ryerson.ca)
  */

#ifndef NAND_H
#define NAND_H
#include "wire.h"
#include "block.h"
/** "newNand" creates and returns a Block_t that implements a NAND gate
  * with the specified name, delay, inputs and output.
  *
  * @param name---The name of the block or NULL or the empty string.
  *      The name may not begin with "0x".
  * @param delay---The propagation delay of the gate.
  * @param w1---the first input wire.
```

```
  * @param the remainder of the arguments are wires. The last argument must
  *    be NULL.  All the wires are inputs, except for the last one which
  *    is the gate's output wire.
  * @return---the newly created block.
  */
extern Block_t newNand(char * name,  int delay, Wire_t w1, ...);
#endif /* NAND_H */
```

E.6.7 event.h

```
/* Copyright (C) 1999 Ken Clowes (kclowes@ee.ryerson.ca) */

/**REVISION HISTORY:
  * Created: March 13, 1999 (kclowes@ee.ryerson.ca)
  */

#ifndef EVENT_H
#define EVENT_H
#include "simulate.h"
#include "wire.h"
#include "value.h"


extern Event_t newEvent(Wire_t w, Value_t v, long time);

extern Wire_t getEventWire(Event_t e);

extern long getEventTime(Event_t e);
extern void setEventTime(Event_t e, long t);
extern Value_t getEventValue(Event_t e);


#endif /* EVENT_H */
```

E.6.8 Event Q implementation

E.6.9 eventQ.h

```
/* Copyright (C) 1999 Ken Clowes (kclowes@ee.ryerson.ca) */

/**REVISION HISTORY:
  * Created: March 13, 1999 (kclowes@ee.ryerson.ca)
  */

#ifndef EVENTQ_H
#define EVENTQ_H
#include "simulate.h"
#include "event.h"
#include "priorityQ.h"

extern EventQ_t newEventQ(void);

extern Event_t removeNextEvent(EventQ_t e);

extern void addEventToQ(EventQ_t eQ, Event_t ev);

extern unsigned int eventQSize(EventQ_t eQ);

#endif /* EVENTQ_H */
```

E.6.10 priorityQ.h

```
/* Copyright (C) 1999 Ken Clowes (kclowes@ee.ryerson.ca) */

/**REVISION HISTORY:
  * Created: March 13, 1999 (kclowes@ee.ryerson.ca)
  */

#ifndef PRIORITYQ_H
#define PRIORITYQ_H
#include "wire.h"
#include "value.h"
```

```
PriorityQ_t newPriorityQ(void);
/** "removeMaxFromPQ" returns the event with largest time value
  * from the priority queue, or NULL if the queue is empty.
  */
extern Event_t removeMaxFromPQ(PriorityQ_t pq);

extern void addToPQ(PriorityQ_t pq, Event_t e);

#endif /* PRIORITYQ_H */
```

E.7 Data Structs and pointers (Appendix B)

Source code for Appendix B. This source code can also be found in the directory `src/datStructsAndPtrs`.

E.7.1 README

```
This directory contains the files associated with
Appendix B---Data Structures and Pointers---in the book "Engineering
Algorithms and Data Structures".

Makefile -- the makefile (what else)
celestialBodies.c  -- Simple example of linked structs
nameDS.c -- How to initialize data strings from strings
            read from stdin.
```

E.7.2 Makefile

```
CFLAGS=-Wall -g -ansi -pedantic
EXECS=nameDS celestialBodies
CC=gcc

all: ${EXECS}

nameDS: nameDS.o
```

```
gcc -o nameDS nameDS.c

celestialBodies: celestialBodies.o
gcc -o celestialBodies celestialBodies.c



clean:
-@rm -f *~ *.dvi *.log *.ps *.log *.aux *.o ${EXECS} *.exe
```

E.7.3 celestialBodies.c

```
#include <stdlib.h>
#include <stdio.h>
#include <assert.h>

typedef struct Body Body, * BodyP;
struct Body {
   char * name;
   BodyP orbits;
};

Body probe, jupiter, earth, moon, sol,
     europa, galacticCentre;

Body probe          = {"Probe", &europa};
Body jupiter        = {"Jupiter", &sol};
Body earth          = {"Earth", &sol};
Body moon           = {"Moon", &earth};
Body sol            = {"Sun", &galacticCentre};
Body europa         = {"Europa", &jupiter};
Body galacticCentre = {"Black hole", NULL};


void orbits(BodyP b)
{
   assert(b != NULL);
```

```
   do {
      printf("%s orbits ", b->name);
      b = b->orbits;
   } while (b != NULL);
   printf("nothing.\n");
}

int main()
{
  orbits(&probe);
  exit(0);
}
```

E.7.4 nameDS.c

```
#include <stdlib.h>
#include <stdio.h>
#include <assert.h>
/* ** typedefs */
typedef struct Name_str Name;

struct Name_str {
   char * first;
   char * last;
};

#define MAX_LEN 128

enum {MAX_DB_SIZE=20};

int main(int argc, char * argv[])
{
  char buf1[MAX_LEN];
  char buf2[MAX_LEN];
  int i, j;
  Name names[MAX_DB_SIZE];

  for(i = 0; scanf("%s %s\n", buf1, buf2) > 0; i++){
```

```
    printf("First: %s   Last: %s\n", buf1, buf2);
    names[i].first = malloc(strlen(buf1)+1);
    names[i].last = malloc(strlen(buf2)+1);
    assert(names[i].first && names[i].last);
    strcpy(names[i].first, buf1);
    strcpy(names[i].last, buf2);
  }
  for(j = 0; j < i; j++)
    printf("Names[%d]: %s, %s\n", j, names[j].last, names[j].first);
  exit(0);
}
```

Bibliography

[Ben99] Jon Bentley. *Programming Pearls*. Addison-Wesley, Reading, Massachusetts, 1999. Paperback, 256 pages.

[FP95] Jr. Brooks Frederick P. *The Mythical Man-Month: Essays on Software Engineering*. Addison-Wesley, Reading, Massachusetts, July 1995. 322 pages.

[GA97] James Gosling and Ken Arnold. *The Java Programming Language*. Addison-Wesley, Reading, Massachusetts, second edition, 1997.

[Hay99] Brian Hayes. Clock of ages. *The Sciences*, November 1999.

[HJ91] Samuel P. Harbison and Guy L. Steele Jr. *C, A Reference Manual*. Prentice-Hall, Toronto, 1991. 392 pages.

[Knu86] Donald Ervin Knuth. *The TeXbook: Computers & Typesetting*. Addison-Wesley, Reading, Massachusetts, 1986. 483 pages.

[Knu97a] Donald Ervin Knuth. *The Art Of Computer Programming (3 volumes)*. Addison-Wesley, Reading, Massachusetts, 1997.

[Knu97b] Donald Ervin Knuth. *The Art Of Computer Programming (Vol 1):Fundamental Algorithms*. Addison-Wesley, Reading, Massachusetts, 1997. 650 pages.

[KP99] Brian W. Kernighan and Rob Pike. *The Practice of Programming*. Addison-Wesley, Reading, Massachusetts, 1999. 267 pages.

[KR88] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice-Hall, Toronto, 2 edition, 1988. 272 pages.

[Lam86] Leslie Lamport. *LaTeX: A Document Preparation System.* Addison-Wesley, Reading, Massachusetts, 1986. 242 pages.

[Nor90] Donald A. Norman. *The Design of Everyday Things.* Doubleday Books, 1990. 257 pages.

[Pik] Rob Pike. *Notes on Programming in C.* 5 pages.

[Pin94] Steven Pinker. *The Language Instinct.* Morrow, New York, 1994.

[THC90] Ronald L. Rivest Thomas H. Cormen, Charles E. Leiserson. *Introduction to Algorithms.* The MIT Press, Cambridge, Massachusetts, 1990. 1028 pages.

Index

Colophon

The book was written using computers running various operating systems (Solaris, Linux, and Windows9x) and the following general and freely available tools:

emacs (or xemacs): The "world's best editor" was used for typing in all the text and programs.

LaTeX: The LaTeX2e[Lam86] document preparation system (based on Donald Knuth's[Knu86] TeX system) was used to format the book. The *bibtex* and *makeindex* utility packages were used to generate the bibliography and index.

latex2html: This package was used to convert the LaTeX2e source code to HTML.

xfig: This package was used to draw the figures.

gcc: The GNU C/C++ compiler was used on all operating systems to compile the C source code.

make: The *make* utility program was used to keep everything up-to-date.

The UNIX command line environment (using the *bash* shell, the *make* utility, the *gcc* compiler, etc.) were used under Windows95 with the freely-availaible `cygwin` package from `www.cygnus.com`.

DRAFT April 8, 2004