S-Q Course Book

Daniel J. Ellard

July 22, 1997

Contents

Chapter 1

Course Information

1.1 Course Guidelines

1.1.1 Grading

Your grade for this course will be based on your numeric score on the homework assignments, hourlies, and the final exam.

The grades will be weighted as follows:

- 50% - five homework assignments (10% each).

  Note that there is an optional sixth assignment; see below for more information.

- 20% - two hourly tests (10% each).

- 30% - one final exam.

The hourlies and the exam are **closed book**: no notes, computers, calculators, or other aids will be permitted. Relevant definitions and formulas will be provided for you on the test.

Assignments represent individual work; no collaboration is permitted. All questions about the assignments should be directed to the course staff; discussion between students about the specific problems on the assignments is not allowed. (Discussion about the course material, however, is strongly encouraged. Study groups are a wonderful way to master the material.)

An optional sixth "make-up" assignment will be made available to any students who specifically request it. The grade for this assignment will replace the lowest grade of the five ordinary assignments, and provides an opportunity for students who do poorly on one of the assignments to demonstrate that they have mastered the material and improve their grade. Note that it is possible that doing the "make-up" assignment will *lower* your grade– if your score on the sixth assignment is lower than the lowest score of your other five assignments, then your cumulative score will also be lower!

1.1.2 Lateness Policy

The course works best when homework is handed in on time– it can be graded and returned promptly (giving you valuable feedback for your next assignment), solution sets can be distributed, and I can get a feel for how well you have learned the material. However, I realize that sometimes this is not possible, due to other commitments you might have, or personal emergencies that might arise.

- Requests for an extension should be made directly to the professor, either in person or via email (ellard@deas.harvard.edu) or telephone (496-6434 during the day, 643-9644 at night). Except in the case of extreme emergencies, requests for an extension must be made at or before the time that the assignment is due.

- There is a deduction of 10% per day late. Late days are not subdivisible; one minute late is the same as one day.

- There is a "grace period" of three days that can be used at any time during the semester. This can be subdivided among two or three assignments, or used all at once. Simply send email to your TF when you submit your assignment, saying that you are using your grace days, and the lateness penalty will not be taken. Once again, late days are not subdivisible; one minute late is the same as an entire day.

- The first two assignments (which are due on Mondays) have a drop-dead date of the following Thursday. The other assignments, which are due on Thursdays, have a drop-dead date of the following Monday. Homework submitted after this time will receive a grade of zero, unless an extension has already been granted.

  Note that the drop-dead dates mentioned earlier are not effected by the grace days.

1.2 Homework Guidelines

In my opinion, the homework assignments are the most important aspect of the course. They are time-consuming and challenging, and should never be left until the last minute.

I have designed the homework assignments to be difficult, but manageable. Some of the problems are straightforward, but others require considerable thought and perhaps a touch of inspiration to solve. I make mistakes, however– sometimes an assignment is much harder or easier than I have anticipated! If you find yourself struggling for days on a single problem, please let me know so I can clarify the problem, restate it, or possibly omit it. At the same time, if you find that there are other problems that you can do effortlessly in a few minutes, let me know so I can challenge you with problems you will find more interesting and educational.

1.2.1 How To Approach the Assignments

- **Start early, and pace yourself.** If you wait until the last minute to begin the assignments, you will regret it.

  The assignments are time-consuming, but not all of that time is productively spent in front of a computer. I personally find that it is most effective for me to read over the assignment as soon as possible, and then ponder each over a period of several days, until all the pieces from lecture and the readings fall into place and I've got a good idea how to solve them, and *then* start coding.

  Starting early also gives you an opportunity to discover that you might need a little extra help from me or a TF to understand some topic related to the assignment.

- **There are no trick questions.**

  If something seems too hard (or too easy), ask about it. There are no *intentionally* misleading or tricky questions, Please let me know if you find any, so I can clarify them.

- **Don't Get Stuck On One Problem.**

  Don't spend all your time working on one very hard problem. Along the same lines, don't spend all of your time gold-plating your answers to the easier problems.

- **Pay Attention To Feedback.**

  In addition to your grade, your TF will give you feedback about your work. Word for word, these are probably the most important things you will read all semester– so study them carefully!

1.2.2 Written Exercises

The assignments usually include several questions that do not require any coding, but instead require written answers. These questions specify the name of the file where you should write your answers.

Please write your answers in clear English or pseudo-code, with tables or diagrams when appropriate or requested. Use complete sentences.

Ordinary text and legible hand-drawn diagrams are sufficient– please do not feel that it is necessary to spend time and energy making your answers into works of art.

1.2.3 Coding Guidelines

1. **Put your name in a comment at the start of each file you edit or change in any way.** In addition, please staple or fasten together securely all the pages of your printouts.

2. **All code must be written in ANSI C, and must compile successfully. Non-compiling code will not be accepted.**

   At this point in your programming career, I don't expect this to be a problem.

3. **All code must compile and run properly on the course (HP-UX) machines.**

   We will *test* and grade your assignments on the course HP-UX machines, so it is in your best interest to test your programs there as well. Code that doesn't work on the HPs is wrong, whether or not it works properly anywhere else.

   In general, I don't care what platform you do your programming on. For some problems, however, we'll give you libraries or sample executables, and unfortunately these libraries and programs usually function properly only on the course machines. For these problems, you *must* use the course machines to compile and test your code.

4. **Don't change the code that is marked** *DO NOT CHANGE.*

   Some of the code that we'll give you should not be changed. Don't change it.

   The code we want you to write or edit should be clearly marked. Look for instructions like **WRITE THIS FUNCTION** in the code or consult the assignment (and when in doubt, ask).

5. **Code to the specifications given, not to the apparent requirements.**

You may notice that, in some cases, some functionality of the code that we ask you to write doesn't seem to ever actually be used by the examples or sample programs– but taking advantage of this fact (either implicitly or explicitly) is a very bad idea, because we grade against the spec.

6. **We do not grade on minor style considerations.** I do reserve the right to deduct points for truly awful code, however.

   In many programming courses, there are deductions for code that doesn't match specific consistency and style guidelines. This course does not have any specific guidelines, except that the grader should be able to figure out how your code works by reading it– it should be abundantly obvious from the organization of the code and the comments you write.

Chapter 2

Weiss Chapter 2 Notes

This course is about algorithms. Before we actually begin our study of any specific algorithms, however, we'll start by studying some of the mathematics necessary to analyze the resources (such as processing time, memory, or disk storage) consumed by the execution of an algorithm.

Two algorithms that solve the same problem may consume wildly different quantities of resources– and this usage may depend upon the inputs to the algorithms, which are not known until the algorithm is actually executed. Depending on the requirements of the system that the algorithm is part of, the quantities of resources used may have a profound effect on the usability of the system, or even whether the system is successful or not.

2.1 Definitions

The *order* of a function $T(N)$ is a description of its rate of growth as N becomes very large.

When analyzing the order of algorithms, the following definitions are commonly used:

- $T(N) = O(f(N))$ if there are positive constants c and n_0 such that $T(N) \leq cf(N)$ when $N \geq n_0$.

  As an example, if $T(N) = 2N + 1$, then $T(N) = O(N)$. To prove this, all we need to do is find c and n_0 such that $2N + 1 \leq cN$ when $N \geq n_O$. If we try $c = 3$, then we can subtract $2N$ from both sides of the inequality and get $1 \leq N$. This is clearly true when $N \geq 1$, so n_0 is 1.

  It may seem strange that in this case $f(N)$ is clearly less than $T(N)$ for all positive values of N, and yet $T(N) = O(f(N))$. The actual values of $T(N)$ and $f(N)$ are not the issue– the important issue is whether a c and n_0 can be found that satisfy this rule.

- $T(N) = \Omega(f(N))$ if there are positive constants c and n_0 such that $T(N) \geq cf(N)$ when $N \geq n_0$.

  This is similar to the big-O notation, but from the other direction– the order of $f(N)$ is typically as large as possible without making it larger than the order of $T(N)$.

  As an example, if $T(N) = 2N + 1$, then $T(N) = O(N)$. To prove this, once again all we need to do is find c and n_0 such that $2N + 1 \geq cN$ when $N \geq n_O$. If we try $c = 2$, then we can subtract $2N$ from both sides of the inequality and get $1 \geq 0$. This is clearly true for any N.

- $T(N) = \Theta(f(N))$ if and only if $T(N) = O(f(N))$ and $T(N) = \Omega(f(N))$.

- $T(N) = o(f(N))$ if and only if $T(N) = O(f(N))$ and $T(N) \neq \Theta(f(N))$.

Remember that in this notation, our goal is to find $f(N)$ (an order), given $T(N)$ (a function). For example, imagine that we have a function $g(N) = 3n^2 + 12$, and we would like to say things about it using big-O notation. It is correct to say "$g(N)$ is $O(n^2)$" but it is usually not correct to say "$O(g(N))$ is n^2".

2.1.1 Big-O Conventions

It may seem strange that the order of $f(N)$ might actually be *greater* than the order of $T(N)$. For example, if $T(N) = O(n^2)$, then it is also true that $T(N) = O(n^3)$, and so on.

By general convention, $f(N)$ is chosen to be the smallest order that is known to be greater or equal to the order of $T(N)$. In this example, you would never say that $T(N) = O(N^3)$ if you knew that $T(N) = O(N^2)$. It would be correct, at least in a mathematical sense, but potentially very misleading to the reader. In fact, in many contexts, big-O is used synonymously with big-theta.

As another convention, any constant factors are dropped from the order. For example, according to the definition given above, it is true that $3n^2 = O(3n^2)$. However, it is also true that $3n^2 = O(n^2)$. The latter form, where the constant factor has been omitted, is the standard way of expressing the big-O of a function.

Similarly, it is also a convention to drop all but the "largest" term of the order. The largest term is the *dominant* term (see section 2.2.1 for more information). For example, $n^2 + 100n + 12 = O(n^2)$, since as n becomes large, the quadratic term will grow more quickly than the others.

2.1.2 Why Not Use Θ All The Time?

At first glance, it would appear that Θ is all we really want or need– after all, it's the "right" answer, where the order of $T(N)$ is exactly $f(N)$. If we know the real order of $T(N)$, then we know $\Theta(f(N))$.

Unfortunately, life is not so simple. Consider the following code snippet:

```
void    strange1 (int A[N])
{
        int     i;

        for (i = 0; i < N; i++) {
                if (A[N] == 0) {
                        break;
                }
                printf ("A [%d] = %d\n", i, A[i]);
        }
}
```

What is the order of `strange1`? It is impossible to know without knowing what the contents of array `A` are: if there is always a zero element in the array, and the first zero element is always at a fixed location in this array (for example, the first location), then this function is $\Theta(1)$– it always takes the same amount of time to run, regardless of N. If we know that there *isn't* a zero element anywhere in the array, then this algorithm is $\Theta(N)$, since it visits every location in the array, and the array is of length N. If we don't know anything about the contents of the array, however, then we can't be sure what the order is, so we don't know what the true order of `strange1`(N) is. We know that the order is *no less* than order 1, so `strange1`$(N) = \Omega(1)$. Similarly, we know that the order is *no larger* than N, so `strange1`$(N) = O(N)$.

2.2 Finding the Big-O of a Function

Although the definitions given above completely define what the big-O of a function is, it is not always immediately obvious how to use them to actually discover the big-O of a function. Finding the order of many useful functions can be a challenging task, and there are even functions that have defied all attempts at a complete analysis! Luckily, since there has been so much work done in the area already, finding the big-O of many functions is simply a matter of finding a known result for a similar function and using it.

The following rules provide a jumping off point:

1. If $T_1(N) = O(f_1(N))$ and $T_2(N) = O(f_2(N))$, then:
   (a) $T_1(N) + T_2(N) = \mathbf{max}(O(f_1(N)), O(f_2(N)))$
   (b) $T_1(N) \times T_2(N) = O(f_1(N) \times f_2(N))$
2. If $T(N)$ is a polynomial of degree k, then $T(N) = \Theta(N^k)$.

   The fact that $T(N) = O(N^k)$ follows from the previous rule. Showing that $T(N) = \Theta(N^k)$ is slightly more complicated, but can be proven by showing that there is always a choice of c and n_0 that satisfies the definition of Θ.
3. $\log^k N = O(N)$, for any constant k.

   This is true, but of relatively limited use. Since $O(N)$ is an upper bound, it is OK to say that anything with a lower order is $O(N)$. This is handy as a last resort, when trying to find the big-O of a nasty function that includes logarithms, but whenever possible it is useful to draw a distinction between $O(N)$ and $O(\log^k N)$. For example, an algorithm that has a big-O of $O(N \log N)$ is (usually) much better than an algorithm that is $O(N^2)$.

2.2.1 The Dominance of Functions

Another method of finding the big-O of a function is to find the dominant term of the function, and find its order. The order of the dominant term will also be the order of the function.

The dominant term is the term that grows most quickly as n becomes large. Some rules of dominance include the following:

- Any exponential function of n dominates any polynomial function of n.
- Any polynomial function of n dominates any logarithmic function of n.
- Any logarithmic function of n dominates a constant term.
- A polynomial of degree k dominates a polynomial of degree l if and only if $k > l$.

There are additional rules that you can discover for yourself. The key is that term $x(n)$ dominates function $y(n)$ if and only if $x(n)/y(n)$ grows as n grows large.

Using these rules can sometimes make finding the order of a complicated function easy. For example, the function $f(n) = n^4 + 2^n + \log n$, is clumsy to manipulate algebraically. However, thanks to the rules of dominance, we know that we the 2^n term will dominate the order of this function, so we can simply find the order of 2^n, which is itself. Therefore, we can immediately say that $f(n) = O(2^n)$.

There are other methods that can be used find the order of functions. but we will not introduce them yet. Later in the semester, when we analyze several recursive algorithms, methods for analyzing the order of recursive functions will be introduced.

2.2.2 Properties of Orders

Since the orders of functions look like ordinary functions, it is tempting to treat them as such, and manipulate them as though they were ordinary functions. **This is usually a huge mistake.**

For example, let $f(n) = n^2$ and $g(n) = 2n^2$. It should be clear that the both $f(n)$ and $g(n)$ are $O(n^2)$. What is the big-O of $g(n) - f(n)$? If we compute the order of the difference as the difference of the orders, then we would mistakenly conclude that $(g(n) - f(n)) =$ the order of $f(n)$ less the order of $g(n)$, or $(n^2 - n^2) = 0$. This answer is obviously wrong. If we actually compute $g(n) - f(n)$, however, we arrive at the correct conclusion: $(g(n) - f(n)) = (2n^2 - n^2) = n^2 = O(n^2)$.

2.3 Big-O Notation and Algorithm Analysis

Now that we have seen the basics of big-O notation, it is time to relate this to the analysis of algorithms.

In our study of algorithms, nearly every function whose order we are interested in finding is a function that defines the quantity of some resource consumed by a particular algorithm in relationship to the parameters of the algorithm. (This function is often referred to as a *complexity* of the algorithm, or less frequently as the *cost function* of the algorithm.) This function is usually not the same as the algorithm itself, but is a property of the algorithm. For example, when we are analyzing an algorithm that multiplies two numbers, the functions we might be interested in are the relationships between the number of digits in each number and the length of time or amount of memory required by the algorithm.

Although big-O notation is a way of describing the order of a function, it is also often meant to represent the time complexity of an algorithm. This is sloppy use of the mathematics, but unfortunately not uncommon. (For example, at the bottom of page 22 in Weiss, a factorial function is described as being $O(N)$. Clearly, the factorial function itself is not $O(N)$, it is $O(N!)$ according to its definition. What is meant is that the particular algorithm given for computing the factorial of N requires $O(N)$ time to execute.) Usually it is clear from the surrounding context what the big-O refers to, but when it is not clear, please ask.

Note that this notation, like the orders themselves, doesn't tell us how quickly or slowly the algorithms actually execute for a given input. This information can be extremely important in practice– during the semester, we will study several algorithms that address the same problems and have the same order running time, but take substantially different amounts of time to execute.

Similarly, the order doesn't tell us how fast an algorithm will run for a small N. This may also be quite important in practice– during the semester, we will study at least one group of algorithms where the choice of the best algorithm depends on N– when N is small, one algorithm is best, but when N is large, a different algorithm is much better.

2.3.1 The Size of a Problem

Big-O notation is used to express an ordering property among functions. In the context of our study of algorithms, the functions are the amount of resources consumed by an algorithm when the algorithm is executed. This function is usually denoted $T(N)$. $T(N)$ is the amount of the resource (usually time or the count of some specific operation) consumed when the input to the algorithm is of size N.

Sometimes it is not obvious what the "size" of the input is, so here are few examples:

- **Multiplication**

  If we multiply two numbers together using the algorithm we learned in grade school, the number of single-digit multiplications and additions that we do depends on the number of digits in the two numbers being multiplied. (If we multiply two numbers together using a calculator, then the number of keystrokes we need to type also depends on the number of digits in the two numbers.) Therefore, for an algorithm that performs multiplication (or any other arithmetic operation) the "size" of the input is usually the number of digits in the input numbers.

  (As an aside, remember that most modern computers can perform arithmetic operations on reasonably-sized numbers (i.e. numbers that are 32 or 64 bits in length) in a single instruction, regardless of the number of non-zero digits in the numbers. Therefore, for most purposes, we will assume that arithmetic operations take a constant amount of time. There are many important applications that deal with numbers large enough so that this is not the case, however.)

- **Searching**

  If we have a set of N unknown items, arranged in an unknown order, and we want to do something involving all of them, then the size of the problem is N. For example, if we want to find the smallest element in an array of N numbers, then the size of the problem is N.

2.3.2 What Does It All Mean?

The big-O complexity of an algorithm is important, but it does not tell the entire story. Any honest and complete analysis of an algorithm should attempt to address the question of what the real cost of executing the algorithm is for real or typical problems.

For example, imagine that (through some some wonderful twist of fate), you must choose between job offers from two companies. The first company offers a contract that will double your salary every five years. The second company offers you a contract that gives you a raise of \$1000 per year. Your salary at the first company increases at $O(2^n)$, while your salary at the second company increases at a rate of $O(n)$. Which position would you choose?

From this information, it is impossible to say. If your starting salaries are the same at the two companies, then what the starting salary is will make all the difference (and if the first company gives you a starting salary of \$0, then you are really in trouble!). If the starting salary at the second company is much higher than the first company, then even the wonderful raises the first company offers might not actually make any difference before you retire. In this case, n isn't going to get too large (assuming an ordinary career), so the behavior of your salary as you become infinitely old is not an issue.

Chapter 3

Sparse Arrays

This chapter explores sparse arrays. A *sparse array* is an array where nearly all of the elements have the same value (usually zero), and this value is a constant, known ahead of time.

In many fields of mathematics and engineering, sparse arrays (in particular, arrays that consist almost entirely of zeros) are quite common. Therefore, it is worthwhile to develop efficient methods for handling such arrays.

In this chapter, we'll explore one way to efficiently represent sparse arrays, and in chapter 4 we'll see how this representation can pay huge dividends, both in terms of space and time.

Note that the examples given in this chapter are all for two-dimensional arrays. However, it is possible to generalize the ideas presented here for arrays of higher dimension; this is left as an exercise.

3.1 A Troublesome Property of C Arrays

In C (and many other languages), space for arrays is typically allocated ahead of time, during compilation, and enough space is allocated to represent *all* of the elements in the array. For example, consider the following variable definition:

```
double bigArray[1000][1000];
```

Defining `bigArray` allocates enough space to hold 1,000,000 doubles. On the HP workstations used by this course, each `double` requires 8 bytes of memory, so this array requires 8 million bytes of memory. Even if we only use a few elements of this array, defining this variable will soak up all of this space. This is OK if we are going to actually use all of this space– but in many cases, we can do a lot better.

C's allocation method is very wasteful in the case of sparse arrays, where we know what most of the values are going to be ahead of time– it is quite wasteful to allocate vast amounts of memory to hold data that we'll never change and whose value we already know!

3.2 Linked List Representation

As we have already seen, one method for representing a 1-dimensional array is as a linked list. We can extend this representation to deal with 2-dimensional arrays by treating each 2-dimensional array as a 1-dimensional array of rows, where each row is in turn a 1-dimensional array. Each "row" array can be implemented as an ordered linked list (where the order is determined by the column of each element). If an element does not appear in the linked list, then it is assumed to have the default value.

Consider the 4-by-4 array illustrated in figure 3.1. This array consists of 12 zero elements, and 4 non-zero elements. Compared to most sparse arrays, this has a relatively high proportion of non-zero elements, but it will serve to illustrate the principles involved.

Figure 3.1: An ordinary C array.

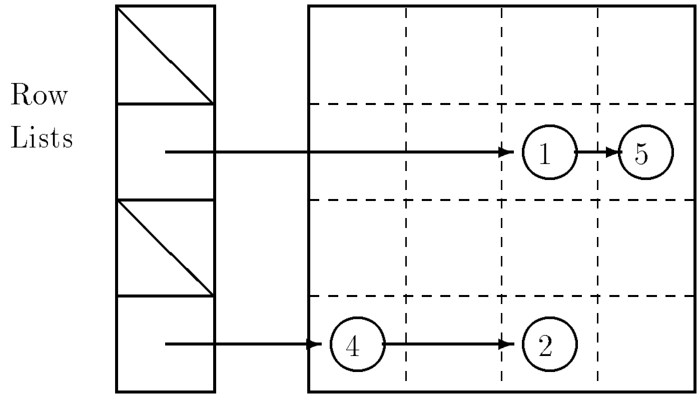
0	0	0	0
0	0	1	5
0	0	0	0
4	0	2	0

An illustration of a sparse array that represents the same array as the array in figure 3.1 is shown in figure 3.2.

The method of accessing elements in a sparse array implemented this way is very simple: find the row list for the row of the element, and search it for the elements in the correct column. If the element is not present, then we simply assume that the value of that element is zero.

Depending on how we access the elements of the array, it may be more appropriate to organize the array by row or by column. For example, if we represent our sparse array

Figure 3.2: A simple sparse array.



with an array of rows (where each row is a linked list), then determining whether or not a particular row is entirely empty is an $O(1)$ operation– either the pointer in the row array is NULL, or it isn't. Determining whether or not a particular column is empty, however, requires checking each row for an element in that column, a potentially $O(n)$ operation.

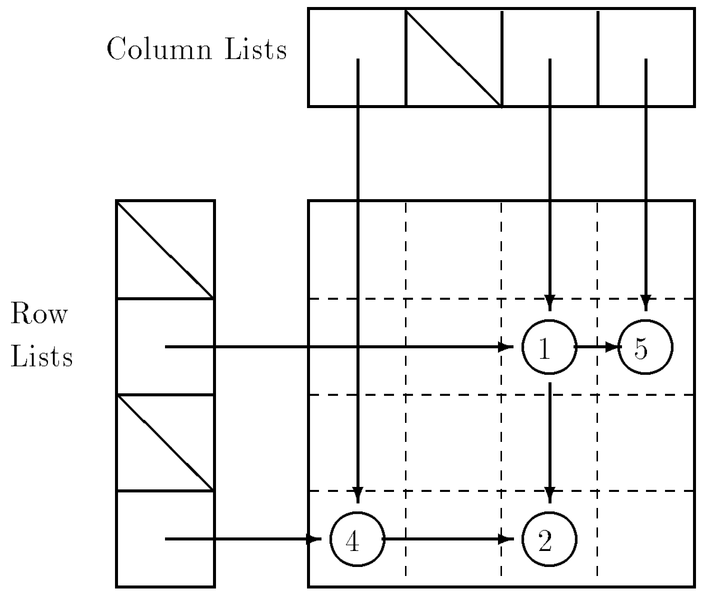
Of course, it is possible to represent the array using both methods: an array of row lists, and an array of column lists. An example of this is illustrated in figure 3.3, which once again represents the same array as in figure 3.2.

Deciding which representation to use is an important issue. Having both row and column lists makes it easy to traverse the array in both row and column order (and as we'll see in chapter 4, this can be quite useful), but it makes other operations more difficult. For example, if row lists are stored, then exchanging two rows of an array[1] is trivial (simply exchanging the pointers to the first element in each row list is sufficient). Storing both row and column lists makes swapping rows more difficult, however, since exchanging two rows may defeat the ordering of the columns.

As always, knowing how the array will be used is essential to picking the optimal representation.

[1] Row exchange is an essential part of some important mathematical algorithms, such as Gaussian Elimination. We won't study these algorithms in this course, but it is useful to remember that row exchange is something that can be important.

Figure 3.3: A sparse array that uses both row and column lists.



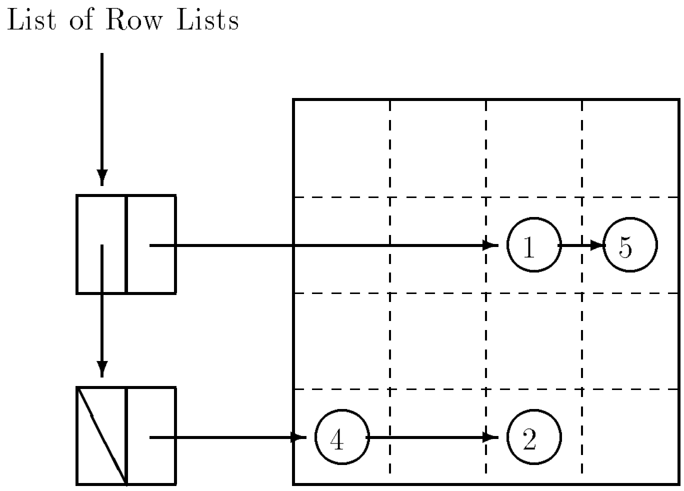
3.2.1 Extremely Sparse Arrays

If most of the rows of the sparse array do not contain any elements, then the array of pointers to row lists is itself a sparse array, and can also be implemented as a linked list. An illustration of this is shown in figure 3.4, which represents the same arrays as figure 3.2, but using a linked list to represent the row array.

From an implementation perspective, adding this is slightly more clumsy than it might appear, because the linked list that represents the row array is a linked list of linked lists, while the linked list that represents each row itself is a linked list of whatever type of element the array actually represents. This means that (at least in a language like C), the linked list routines we use for the rows can't be used for the row array, because of this type mismatch.

If the number of rows that actually contain any elements is small, then this approach can offer large space savings– but if many of the rows are occupied, then as much space can be consumed by this representation as the array. In addition, the access time is considerably slower, because we've replaced an $O(1)$ array lookup with a (potentially) $O(n)$ linked list traversal.

Figure 3.4: A sparse array where the array of rows is replaced by a linked list.



3.3 An Implementation

The code in `sa.c` (with header file `sa.h`) provides a relatively simple and straightforward implementation of sparse arrays. (The entire source for this library is available online, and is not included here.)

Note that the representation that `sa` uses includes both row and column lists; it could be somewhat simpler if only rows or columns were used.

Also, as an extra feature, the default value of the elements in the array is not assumed to be zero. Instead, this value can be specified when the array is created.

```
/*
 * An "sa_cell_t" is a simple singly-linked list
 * structure.  Each cell represents an element of
 * the array, and each list represents a single
 * row or column of the array.
 */

typedef struct _sa_cell_t {
        struct _sa_cell_t *next; /* pointer to next.    */
```

```
        int     index;          /* index in row or col. */
        int     value;          /* value of the elem.   */
} sa_cell_t;

/*
 * An sa_t implements a sparse array.  Note that
 * this implementation includes both row and column
 * lists.
 */

typedef struct  {
        int     num_rows;       /* # of rows in array.  */
        int     num_cols;       /* # of columns.        */
        sa_cell_t **rows;       /* array of row lists.  */
        sa_cell_t **cols;       /* array of col lists.  */
        int     default_val;    /* default elem value.  */
} sa_t;

```

3.3.1 An Optimization: Taking Advantage of Order

Consider the problem of displaying the contents of a sparse array by printing them on the screen. The obvious algorithm is to print the array in the ordinary manner, entirely ignoring the fact that the array is sparse. An implementation of this algorithm is given by function `saPrint`, shown below.

```
void            saPrint (sa_t *sa)
{
        int             i, j;

        for (i = 0; i < sa->num_rows; i++) {
                for (j = 0; j < sa->num_cols; j++) {
                        printf ("%4d ", saGet (sa, i, j));
                }
                printf ("\n");
        }
}
```

Ideally, the cost of printing an n-by-n array is $O(n^2)$, since every element in the array must be printed, and there are n^2 elements. Unfortunately, this method is inefficient because of the cost of the `saGet` function (which is used to fetch the value of each element in the array). In the worst case, `saGet` is $O(n)$, so `saPrint` has a worst-case running time of $O(n^3)$.

However, we can take advantage of the ordered property of the lists in order to speed this up quite a bit. Since the order we wish to print the elements in is the same as the order that they appear in the lists, there is no reason to start over at the head of the list (as `saGet` must do) each time we need to access an element. Instead, we can keep track of the location of the last element that we visited, and begin our search there.

Alternatively, we can take even more advantage of the fact that the row lists are kept in order. We can simply traverse each row list, looking ahead to see where the next non-zero element is and printing out the appropriate number of zeros between. This principle is illustrated in `saPrintFast`.

```
void		saPrintFast (sa_t *sa)
{
	int		i, j;
	sa_cell_t	*cursor;

	for (i = 0; i < sa->num_rows; i++) {
		cursor = sa->rows [i];
		for (j = 0; j < sa->num_cols; j++) {
			if ((cursor == NULL) || (cursor->index > j)) {
				printf ("%4d", sa->default_val);
			}
			else {
				printf ("%4d", cursor->value);
				cursor = cursor->next;
			}
			printf (" ");
		}

		printf ("\n");
	}

	return ;
}
```

3.3.2 Cursors

In `saPrintFast`, `cursor` is used to keep track of where we are in the list. In general, a *cursor* is a way to keep track of an "interesting" location in a data structure, so that this location doesn't need to be recomputed again. In this case, we keep track of where we are in the list so we don't have to traverse to the same spot again.

The notion of a cursor is also mentioned in Weiss, section 3.2.8, in a somewhat different context– but the idea is the same. Instead of searching the `CursorSpace` array for the index of the unused element, the index of an available element (if there are any) is always stored in the first location of the array.

3.4 How Space-Efficient Are Sparse Arrays?

In the implementation shown in `sa.c`, on the HP workstations (which use 4-byte integers and 4-byte pointers) the space requirements of an n by n sparse array containing m non-zero elements are:

- `sizeof (sa_t)` = 20 bytes.
- `2 * sizeof (sa_cell_t *) * n` = $8n$ bytes ($4m$ each for the `rows` and `cols` arrays).
- `2 * sizeof (sa_cell_t) * m` = $24m$ bytes for the m values.
- The `malloc` library may need to allocate some extra space for its own use.

As we can see, this representation is not worthwhile unless the arrays are indeed fairly sparse: each element requires at least 6 times as much memory to represent as it would in an ordinary array. (If only row or column lists are maintained, then the requirements are cut in half, but this may have a profound effect on the running time of algorithms that access the array.)

3.5 A Note About Software Engineering

This course focuses on algorithms, and how they can be implemented. Unfortunately, there isn't time to also focus equally on the software engineering aspects of how these algorithms are implemented in C.

If you have taken a course in software engineering (or have been introduced to these principles in an introductory programming course or from actual work experience), then you probably realized that the implementation of cursors shown above is terrible from the perspective of software engineering. It makes assumptions about the nature of the underlying `sa_t` and `sa_cell_t` structures that should be hidden by a layer of abstraction. A small change to the `sa_cell_t` structure could easily break this code.

In most cases, would be better to implement the idea of cursors as part of the `sa` library itself, so that functions that use this library do not depend on picayune details of the actual structures used by the library. The underlying algorithms will be entirely the same, of course, but the code would be more abstract and more reusable. In this text, however, I will usually sacrifice software engineering principles in favor of clean and concrete algorithms.

3.6 Exercises

1. In the sparse array representation used in this chapter, the linked lists are always kept in order by index. Keeping these lists in order requires a small amount of extra work during insertion, but can also save work during lookup. Discuss the tradeoffs between keeping the lists ordered versus unordered. List any situations you can imagine where one or the other method is troublesome or inefficient.

2. Devise an efficient algorithm that finds the intersection of two ordered linked lists (i.e. the list of values that appear in both lists) and prints it in order. The algorithm must visit each node in both lists *at most* once.

   Hint– consider the use of a cursor.

3. In many contexts, it is appropriate for uninitialized elements of a sparse array to be treated as having an undefined value, instead of having the default value. In such an array, for example, using `saGet` on an element that hadn't been assigned a value with `saSet` would be considered an error. What changes (if any) to the `sa` library would be necessary to implement this behavior?

4. Implement a library, similar to `sa`, that implements three-dimensional sparse arrays.

5. As mentioned in section 3.5, some of the `sa` code could be rewritten in a much better manner, with a cleaner and more abstract interface. Starting with the current `sa` library, define a cleaner interface for sparse arrays, and implement the corresponding library. Be sure to include cursors in your library.

Chapter 4

Matrix Operations

Matrix operations are very useful and common in many areas of engineering and sciences and are computationally expensive– and therefore, they are of interest to computer science.

You may have encountered matrices already, but since knowledge of matrices is not listed as a prerequisite of this course, I do not assume that this is the case. The first few sections of this chapter define what a matrix is (at least in the context of S-Q) and some of the basic matrix operations and their properties. Even if you are already familiar with matrices, you might find it useful to skim these sections in order to familiarize yourself with the notation.

4.1 Terms and Definitions

In general, a *matrix* is a 2-dimensional array, together with some operations, such as multiplication and addition, defined for the array.

For the sake of simplicity, this chapter assumes that all the matrices are n-by-n "square" arrays: the number of rows and columns are the same. Generalizing the ideas to work with rectangular arrays is left as an exercise.

Note: the subscripts or indices of the elements of the matrix range from $(0,0)$ (at the upper left corner of the array) to $(n-1, n-1)$ (at the lower right corner). This is somewhat different than you might see in mathematical texts, where the subscripts start and 1 and end at n. I have made this change in order to make it easier to understand matrices in the context of C's zero-based arrays, but you should be aware that my notation is different than what you might see elsewhere.

For example, the C array written `int A[2][2]` would be written as matrix

$$A = \begin{bmatrix} a_{0,0}, a_{0,1} \\ a_{1,0}, a_{1,1} \end{bmatrix}$$

The matrix element denoted as $a_{i,j}$ corresponds to the C array element `A[i][j]`.

4.2 Matrix Addition and Multiplication

4.2.1 Addition

Addition of two matrices is very simple: the sum of two matrices is the matrix composed of the sums of elements of the two matrices:

$$A + B = \begin{bmatrix} (a_{0,0} + b_{0,0}) & \cdots & (a_{0,n-1} + b_{0,n-1}) \\ \vdots & (a_{i,j} + b_{i,j}) & \vdots \\ (a_{n-1,0} + b_{n-1,0}) & \cdots & (a_{n-1,n-1} + b_{n-1,n-1}) \end{bmatrix}$$

This definition is expressed in C in algorithm 4.1.

Algorithm 4.1 (Matrix Addition) *An algorithm to add two n-by-n matrices A and B, storing the result in n-by-n matrix C.*

```
void    mm_add (int A[N][N], int B[N][N], int C[N][N])
{
        int i, j;

        for (i = 0; i < N; i++) {
                for (j = 0; j < N; j++) {
                        C [i][j] = A [i][j] + B [i][j];
                }
        }
}
```

Algorithm 4.1 is clearly $O(n^2)$.

4.2.2 Multiplication

The algorithm for multiplication is quite a bit different than might be expected– instead of simply multiplying the elements together, each element of the product is the result of several operations, as described in algorithm 4.2.

Algorithm 4.2 (Matrix Multiplication) *An algorithm to multiply two n-by-n matrices A and B, storing the result in matrix C. This algorithm assumes that C is not the same matrix as A or B (if A or B are the same array as C, the wrong result is likely because the contents of C are changed during the calculation).*

```
void    mm_mul (int A[N][N], int B[N][N], int C[N][N])
{
        int i, j, k;
        int sum;

        for (i = 0; i < N; i++) {
                for (j = 0; j < N; j++) {
                        sum = 0;
                        for (k = 0; k < N; k++) {
                                sum += A[i][k] * B[k][j];
                        }
                        C[i][j] = sum;
                }
        }
}
```

Algorithm 4.2 is clearly $O(n^3)$.

4.3 Faster Matrix Multiplication Algorithms

Algorithm 4.2, which defines the operation of matrix multiplication for square matrices, is clearly $O(n^3)$. Since there are many algorithms that use matrix multiplication, often on relatively large matrices, this can have a serious effect on the speed of the programs that use these algorithms. For this reason, many algorithms that use matrix multiplication use various tricks which take advantage of some particular attribute of the specific matrices used by that algorithm. (We'll see an example of one such trick in section 4.4.)

However, for many years it was assumed that general matrix multiplication was an intrinsically $O(n^3)$ operation, so despite the benefits that a faster general multiplication algorithm would have, there was not a lot of research dedicated to finding one. However, in 1969 Strassen discovered an $O(n^{2.81})$ algorithm[1]. This may seem like a small improvement (and in fact, it does not really translate into real savings unless the matrices are large), but it was an astonishing result that spurred a great deal of research in this area. To the best of my knowledge, however, nobody has yet proven what the big-O of the fastest possible general matrix multiplication algorithm is[2]. Perhaps there are some useful and theoretically important discoveries still to be made.

4.4 Operations on Sparse Matrices

As implied earlier, there are tricks that can be used to speed up matrix multiplication if the matrices are known to have particular properties.

One kind of matrix that can be multiplied quickly is a sparse matrix. A *sparse matrix*, like a sparse array, is a matrix where most of the elements are the same value. In most cases, the common value is zero, but this is not necessarily the case. In this section, we'll begin by exploring how sparse matrices that are mostly zero can be multiplied quickly, but at the end we will see how the principles for handling these matrices can also be used to speed up operations on other kinds of sparse matrices.

4.4.1 Multiplying Sparse Matrices

A simple-minded approach to matrix multiplication is to blindly use algorithm 4.2 again, but instead of using the ordinary array operations to access to matrix elements, use the sparse array (or multilist) operations to access them:

```
void    mm_mul_sa (sa_t *A, sa_t *B, sa_t *C)
{
        int i, j, k;
        int sum;
```

[1] Strassen's algorithm is described in section 10.2.4, pages 378-380 of the Weiss book.

[2] Of course, it can't be faster than $O(n^2)$, since an n-by-n matrix has n^2 elements, and each has to be visited once just to fill in the answer– but how close to $O(n^2)$ can we get?

```

        for (i = 0; i < A->num_rows; i++) {
                for (j = 0; j < B->num_cols; j++) {
                        sum = 0;
                        for (k = 0; k < A->num_cols; k++) {
                                sum += saGet (A,i,k) * saGet (B,k,j);
                        }
                        saSet (C,i,j,sum);
                }
        }
}
```

This is a step in the wrong direction, however: since `saGet` and `saSet` are generally $O(n)$ operations themselves, the resulting algorithm is $O(n^4)$ in the worst case.

However, there is a way to speed this up considerably by changing the inner loop (in this code, the `k` loop) to take advantage of two key facts:

- The product of any number and zero is zero. Therefore, if we know that either $A[i][k]$ or $B[k][j]$ is zero, we can skip the multiplication and addition.
- The loop strides through the actual (non-zero) elements in the array in the same order that they appear in the linked lists that are used to represent the array. Therefore, instead of starting at the base of each list (as `saGet` and `saSet` must do), we can keep track of where we are in each list, so that we can find the next element very quickly.

The inner loop of algorithm 4.3 avoids trying each possible k (as algorithm 4.2 did) by moving down both the row and column lists in tandem, searching for elements in the row list that have the same index as elements in the column list. Since the lists are kept in order by index, we know that we can do this in one scan through the lists. Each iteration through the loop moves forward at least one position in one of the lists, so the loop terminates after at most $R + C$ iterations (where R is the number of elements in the row list and C the number of elements in the column list).

4.4.2 Another Note About Software Engineering

Implementing the algorithm this way (i.e. writing this function in a way that depends on so many details of the `sa_t` structure is clearly another abuse of software engineering principles. See section 3.5 for more discussion.

Algorithm 4.3 (Sparse Matrix Multiplication) *An algorithm to multiply two sparse n-by-n matrices A and B, storing the result in matrix C. This algorithm assumes that C is a different matrix than both A and B, and that C is initially empty (all its elements are zero).*

This algorithm is identical to algorithm 4.2, except for the inner block, which is changed as follows:

```
void    mm_mul_sa_fast (sa_t *A, sa_t *B, sa_t *C)
{
        int i, j, sum;
        sa_cell_t *row_p, *col_p;

        for (i = 0; i < A->num_rows; i++) {
                for (j = 0; j < B->num_cols; j++) {
                        row_p = A->rows [i];
                        col_p = B->cols [j];
                        sum = 0;

                        while ((row_p != NULL) && (col_p != NULL)) {
                                if (row_p->index == col_p->index) {
                                        sum += row_p->value * col_p->value;
                                        row_p = row_p->next;
                                        col_p = col_p->next;
                                }
                                else if (row_p->index < col_p->index)
                                        row_p = row_p->next;
                                else
                                        col_p = col_p->next;
                        }
                        saSet (C, i, j, sum);
                }
        }
}
```

4.4.3 Other Sparse Matrices

As promised, we will now extend the techniques we have developed for multiplying sparse matrices where the common value is zero to handle sparse matrices where the common value is some other number.

Let A be a sparse matrix where most of the elements have value α, and B be a sparse matrix where most of the elements have a value of zero. Our goal is compute AB efficiently.

A can be written as the sum of two matrices: one where all of the elements are α, and the other the difference between this α matrix and A:

$$A = \begin{bmatrix} a_{0,0} - \alpha, & \cdots, & a_{0,n-1} - \alpha \\ \vdots & a_{i,j}, -\alpha & \vdots \\ a_{n-1,0} - \alpha, & \cdots, & a_{n-1,n-1} - \alpha \end{bmatrix} + \begin{bmatrix} \alpha, & \cdots, & \alpha \\ \vdots, & \alpha, & \vdots \\ \alpha, & \cdots, & \alpha \end{bmatrix}$$

We'll denote the matrix that results from subtracting α from every element of A as $A_{a-\alpha}$, and the matrix that consists entirely of αs as A_α.

This is useful, because matrix multiplication is distributive:

$$\begin{aligned} X(Y+Z) &= XY + XZ \\ (X+Y)Z &= XZ + YZ \end{aligned}$$

This property follows from the definition of matrix multiplication; the proof is left as an exercise.

However, matrix multiplication is not commutative: in general $XY \neq YX$. Therefore, it is very important to get the order right when we do multiplication!

Therefore, $AB = (A_{a-\alpha} + A_\alpha)B = A_{a-\alpha}B + A_\alpha B$.

At this point, it seems like we're making more work for ourselves, because we are now doing two matrix multiplications and then adding the results together. However, with a little more work we can simplify this calculation enormously.

Since $A_{a-\alpha}$ and B are both sparse matrices with a common value of zero, the product $A_{a-\alpha}B$ can be computed efficiently using the algorithm described in the previous section.

Since every row in A_α is identical, every row in the product of $A_\alpha B$ must also be identical as well. In fact, each element in the result is αn multiplied by the sum of the elements in the corresponding column of B. Therefore, the product $A_\alpha B$ is

$$A_\alpha B = \begin{bmatrix} \alpha n \sum_{j=0}^{j=n-1} b_{0,j}, \cdots, & \alpha n \sum_{j=0}^{j=n-1} b_{i,j}, \cdots, & \alpha n \sum_{j=0}^{j=n-1} b_{n-1,j} \\ \vdots & \vdots & \vdots \\ \alpha n \sum_{j=0}^{j=n-1} b_{0,j}, \cdots, & \alpha n \sum_{j=0}^{j=n-1} b_{i,j}, \cdots, & \alpha n \sum_{j=0}^{j=n-1} b_{n-1,j} \end{bmatrix}$$

The sum for computing each value of the first row takes $O(n)$, but once the first row is finished, each other row is just a copy of the first. Therefore, this multiplication is $O(n^2)$.

Computing the order of the entire calculation depends upon the "sparseness" of A and B. If they are very sparse, and the sparse array multiplication algorithm from the previous section is modified to take full advantage of this (see the exercises for more information), then the multiplication of $A_{a-\alpha}B$ can be done faster than $O(n^2)$, so that the $O(n^2)$ addition at the end of the calculation dominates the order of calculation. If A and B are not so sparse, then this multiplication can be $O(n^3)$.

4.5 Exercises

1. Another important optimization that can be used to improve algorithm 4.3 is to skip over rows or columns that are entirely zero.

   For example, in the current algorithm, even if a row is completely zero, the inner two loops are run over it anyway, even though no non-zero values can possibly result– $O(n^2)$ work that accomplishes nothing. In the very worst case, multiplying two sparse matrices that are both entirely zero will require $O(n^3)$ operations (or $O(n^2)$, if the solution to the previous problem is used).

   Improve the algorithm so that it handles rows or columns that are entirely zero in a more efficient manner. Discuss any tradeoffs that you make between efficiency and simplicity.

2. Extend the ideas in section 4.4.3 to work when B is a sparse array with common value β (where $\beta \neq 0$).

Chapter 5

Tries

A *trie* (pronounced "try") is a data structure that supports efficient FIND, INSERT, and DELETE operations for keys that can be represented by a unique string. In fact, the time complexity of *any* of these operations is $O(1)$ with respect to the number of keys present in the trie. The factor that determines the worst-case time it takes to perform any of these operations on a trie is the length of the key. For example, if we used a trie to implement a telephone directory, it would take the same amount of time to find "Ellard, Dan" whether we were looking in the Harvard telephone book or a telephone book for the entire United States. It could take twice as long to look up "Titmouse, Barneswoggle", however, simply because this name is twice the length of "Ellard, Dan".

5.1 Tries as an Abstract Data Type

As an abstract data type, a trie supports three fundamental operations:

- **FIND**(k)

  Determine whether k is present in the trie. Depending on the application of the ADT, **FIND** may either simply indicate whether or not k is present, or it may return a value v associated with the key k.

- **INSERT**(k, v)

  Add the key k to the trie. Depending on the application, a value v may or may not be supplied.

- **DELETE**(k)

  Remove the key k from the trie.

The domain of k (that is, the set of types that satisfy the constraints on k) is the set of finite strings from a specific alphabet. The domain of v is unrestricted.

The notion of strings used here is more general than the definition of strings in C, and is worth investigating. A *string* is defined to be to be a finite, ordered sequence of elements (called *characters* of a finite set (called the *alphabet*). In C, for example, the alphabet is the set of all possible `char`s, with the exception of the `char` 0, which is used to denote the end of a C string but is not considered part of the string), and each character is simply a non-zero `char`. However, other kinds of strings are quite useful and common (in the context of tries, but also in other contexts). For example, strings of bits (which have an alphabet of $\{0, 1\}$, and each character is a single bit) or strings of decimal numbers are often useful. For all of the examples in this section, we will use strings that have an alphabet consisting of the lowercase letters 'a' through 'z'. In the code samples, we will use the C convention of using a zero (`'\0'`) character to mark the end of a string, but note again that this character is not considered to be part of the string itself.

Note that the keys must be unique; while it is relatively easy to adapt most of the data structures that we have seen so far to work with multiple elements that have the same key, the same is not true for tries.[1]

As we will see later, most implementations of tries work best when the alphabet of their strings is small, although from the perspective of the ADT, there is no limitation placed on the size of the alphabet, as long as it is finite.

Tries also readily support some other operations, which will be discussed in section 5.6.

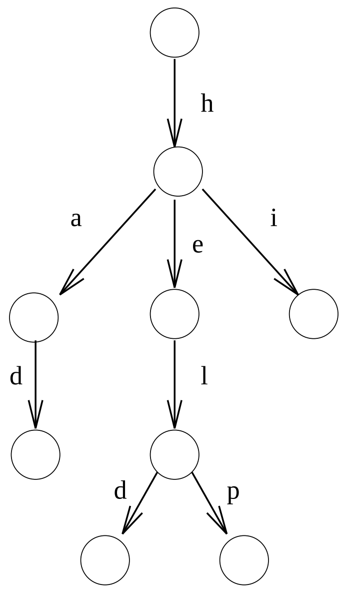
5.2 Tries as a Data Structure

As we have seen, when we search for a key in a binary search tree, at each node we visit, we compare the key in that node to the key that we are searching for. All that we care about is whether the key in the node is less than, equal to, or greater than the key we are searching for, however– the precise value of the node's key is not particularly interesting, unless it is an *exact* match.

If you think carefully about it, you will realize that this manner of search potentially throws away quite a bit of information at each comparison. The path from the root to any

[1]It is possible to represent duplicate keys in a trie, but the result is a more complicated data structure, whose design is left as an exercise.

Figure 5.1: A simple trie containing the keys “had”, “held”, “help”, and “hi”.



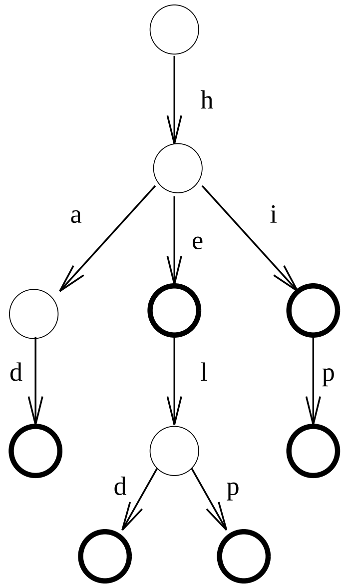
node in a binary search tree may represent a considerable amount of information about the contents of the node (and may, in some cases, specify it completely).

A *trie* is a data structure that stores the information about the contents of each node in the path from the root to the node, rather than the node itself. For example, consider the tree shown in figure 5.1. In this tree, each path between the root and a leaf represents a key, so this trie contains the keys “had”, “held”, “help”, and “hi”. Note that the nodes themselves are unlabeled. Instead, each *transition* between two nodes is labeled with a single character from a key.

However, this manner of representing keys has a fatal flaw; it is impossible to represent any key that is a prefix of another. This problem is fixed by introducing a special marker on each node that tells whether or not it represents the end of a key. In this text, a node represented by a light circle is an “internal” node, while a node represented by a dark circle represents the end of a key. In the code that is used to express the algorithms in this section, a node that represents the end of a key has a non-zero value in its `endOfKey` field. The trie shown in figure 5.2 contains the same keys as the trie in figure 5.1, with the addition of “he” and “hip”.

The algorithm for determining whether or not a particular word is present in the trie is very simple, and is given in algorithm 5.1.

Figure 5.2: A trie containing the keys “had”, “he”, “held”, “help”, “hi”, and “hip”.



If the trie is used to store information about the keys (in addition to the keys themselves), then algorithm 5.1 can be modified to return this information instead of simply FOUND and NOT_FOUND.

The algorithm for inserting a new key is more involved. Like the algorithm for FIND, it traverses the trie until it either reaches the end of the string, or it discovers that the trie does not contain the string. In the first case, all that is necessary is to mark the end node as being the end of a key (and depositing a value into this node, if appropriate). In the second case, the trie must be extended with new nodes that represent the remainder of the string. A recursive algorithm for INSERT is shown in algorithm 5.2.

The algorithm for deleting a key is similar to insert, but has more special cases to consider. Like insert, it traverses the tree until the end of the key string is reached. If the end node has children, then this node cannot be removed, but all that needs to be done is to remove the marker that tells that it is the end of a key. If the node does not have any children, however, then it can be removed, as can the entire sequence of single-child, non-end-of-key nodes that lead to this node.

For example, in figure 5.3, deleting “are” causes nodes 3 and 4 to be deleted, but not 1 and 2 (because these represent part of the other keys represented by the trie). Deleting “at” from the original trie cannot delete any nodes, but simply removes the `endOfKey` marker on

Algorithm 5.1 (Trie FIND) *An algorithm to determine whether a specified key is present in a trie. Returns FOUND if the key is in the trie, NOT_FOUND otherwise.*

```
int trieFind (trie_t *root, char *key)
{
        int     i       = 0;
        trie_t  *curr   = root;
        int     key_len = strlen (key);

        if (root == NULL)
                return (NOT_FOUND);

        for (i = 0; i < key_len; i++) {
                curr = trieFindChild (curr, key[i]);
                if (curr == NULL)
                        return (NOT_FOUND);
        }

        if (trieIsEndOfKey (curr))
                return (FOUND);
        else
                return (NOT_FOUND);
}
```

Algorithm 5.2 (Trie INSERT) *An algorithm to insert a key into a trie.*

```
trie_t *trieInsert (trie_t *root, char *key)
{

        if (root == NULL)
                root = trieCreate ();

        if (key[0] == '\0') {
                trieEndOfKey (root, END_OF_KEY);
        }
        else {
                trie_t *child = trieInsert (trieFindChild (root, *key),
                                key + 1);
                trieInsertChild (root, key[0], child);
        }

        return (root);
}
```

Figure 5.3: A trie containing the keys “are”, “at”, and “ate”.

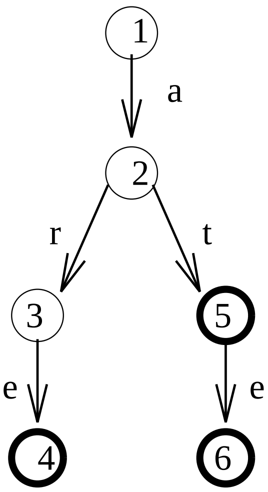


Figure 5.4: A 26-ary tree node to implement a trie of strings of lowercase letters.

```
typedef struct  _trie_t {
        struct  _trie_t *children[MAX_CHILDREN];
        int             value;
        int             endOfKey;
} trie_t;
```

node 5. Deleting “ate” from the original trie removes node 6, but cannot remove node 5 (even though node 5 has only a single child), because node 5 represents the end of a key.

5.3 Implementing Tries

The most obvious way of implementing a trie is as an n-ary tree, where n is the number of elements in the alphabet. In addition to the child pointers in each node, each node also contains a flag that indicates whether or not it is an end-of-key node, and the value associated with the key (if any). For example, if we wanted to construct a trie for the alphabet of the 26 lowercase letters, where each key has an integer associated with it, the C definition shown in figure 5.4 for the `trie_t` structure could be used.

Each letter of the alphabet maps to one of the indices of the `children` array, and writing the code to navigate through the resulting tree is quite simple.

However, from an engineering point of view, this simplistic approach can be disastrous for many applications, because the typical number of children of each node is far fewer than the size of the alphabet. For example, in figure 5.2, each node has, on average, 0.9 children, although this representation must always allocate enough space for each node to have 26 children.

For example, on the HP computers, `sizeof(trie_t) == 112`, so if we used this structure to represent the six keys in the trie illustrated in figure 5.2, this structure would require 1120 bytes of memory to represent six strings which have a combined length of only 24 bytes!

As another example, a trie consisting of the 20,000 lower-case words in my computer's small spell-checking dictionary requires 62,000 nodes to represent, for a total of nearly 7 megabytes of memory, even though the dictionary itself is only 200,000 bytes in length. Thus, a trie to represent this small dictionary requires nearly 35 times more memory than the dictionary itself.

Seven megabytes might not seem like a very large amount of memory by today's standards, but note the factor of 35 increase in memory consumption– by picking a larger problem, we can soon soak up enough memory to cause concern on any system. Depending on the nature of the keys represented by the trie, this factor may grow or shrink, but a few empirical experiments suggest that this factor is actually *lower* than many typical sets of keys!

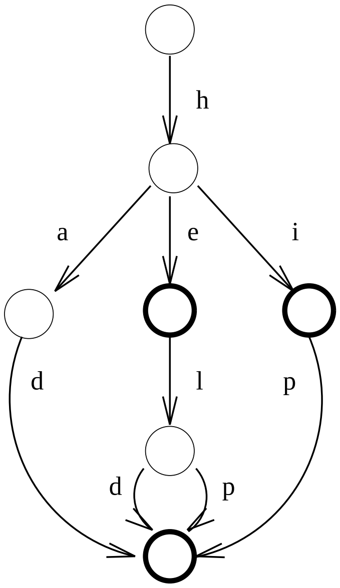
If we expand our representation to allow uppercase letters and punctuation, then the memory requirements would increase even further. However, this might not be necessary, at least in the case of the dictionary; only a small percentage of the words in the dictionary (at least, the spell-checking dictionary I have) contain uppercase letters, and nearly all of these uppercase letters appear as the first letter in a word. Therefore, it would be possible to handle nearly all words that contain an uppercase letter by adding a special case to the way that the root node of our trie is treated. Punctuation is more troublesome, however, since the position of punctuation in words is not as conveniently localized.

5.4 Reducing Memory Requirements: Eliminating Nodes

5.4.1 Special Case: Leaf Nodes

One observation is that in our implementation, leaf nodes are particularly wasteful: leaf nodes have no children, but require as much space as nodes that have 26 children.

Figure 5.5: A trie with a single shared leaf.



If the only thing that we are interested in storing in the trie is a set of keys (with no associated values), then leaf nodes contain no useful information except that their `endOfKey` value is non-zero. This suggests an immediate optimization: represent *all* of the leaf nodes with a single leaf. This idea is illustrated in figure 5.5, which once again represents the same keys as the trie in figure 5.2.

If we do wish to associate values with each key, then unique leaf nodes are necessary– but they only need to be large enough to hold the value, and can omit the pointers to 26 children.

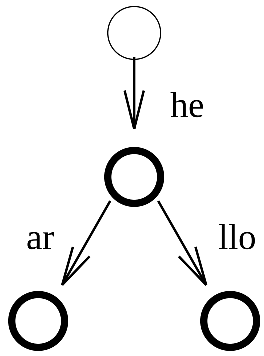
5.4.2 Patricia Tries

One variation of tries is called a *Patricia Tree* (or simply *Patricia*).[2] The actual patricia data structure is somewhat different from what I describe here; I am focusing on a specific idea used by patricias that is particularly useful in our effort to reduce the memory consumption of tries. See Knuth, *The Art of Computer Programming, vol 3*, or Lewis and Denenberg, *Data Structures and Their Algorithms* for a full exposition of patricias.

In a patricia, sequences of nodes that all have only one child and do not represent the

[2] Patricia stands for “Practical Algorithm To Retrieve Information Coded in Alphanumeric”.

Figure 5.6: A patricia representing “he”, “hello”, and “hear”.



end of a key are collapsed into a single transition. We can apply this idea to our tries by adding the possibility that each transition represents several characters of a string, instead of just one. An example of such a trie, representing the keys “he”, “hello”, and “hear” is shown in figure 5.6. Note that if only keys are stored in this kind of trie, then the trick of combining leaf nodes can be used again here, although this is not illustrated in figure 5.6.

At first glance, this would seem to greatly complicate the search algorithm, since during a search, at each node we might have to look at an unknown number of characters of the key we are searching for instead of single character. However, in actuality this does not complicate matters very much. Since only sequences of nodes that have one child are collapsed, no such collapsed sequence can possibly be a prefix of another. Therefore, although each transition might represent many characters, only one transition (the transition that begins with the next character in the key) can possibly be valid at each step. Therefore, the search is straightforward and efficient.

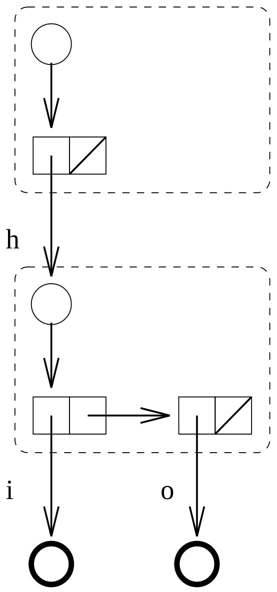
INSERT and DELETE in a trie constructed this way is slightly more complicated. When a key is deleted, it might present an opportunity to collapse a sequence of transitions, and when a key is added it may be necessary to uncollapse a transition.

5.5 Reducing Memory Requirements: Smaller Nodes

There are two ways to reduce the memory consumption of a trie– reduce the number of nodes (as the techniques mentioned above attempt to do), or reduce the amount of memory required by each node. In this section, we will explore a technique that attempts to do the latter.

The problem is that the array of pointers to children is typically very sparse. We have

Figure 5.7: A de la Briandais tree containing “hi” and “ho”.



seen sparse arrays before, so we know an approach to this problem: instead of storing the array explicitly as an ordinary C array, we can store the array as a linked list. This technique is generally called a *de la Briandais tree* (after Rene de la Briandais, who developed many of the ideas behind the practical implementation and application of tries). An example of a de la Briandais tree is shown in figure 5.7.

Traversing a linked list to find a child pointer (or discover that there is not one) is a considerably more expensive operation than a simple array lookup, so tries implemented in this manner are not as time-efficient as our array-based tries. The decrease in speed is proportional, in the worst case, to the size of the character set in use– for our example application, where there are 26 possible characters, we might need to examine as many as 25 linked list elements at each node before we find the child we are looking for, or determine that it does not exist. However, this slowdown is proportional *in the worst case* to the size of the alphabet, and is not related in any way to the number of keys in the trie or their lengths.

Nodes that have many children may actually require more memory to represent using linked lists than arrays, since the linked list cells themselves require more memory than the array elements, but if the trie is sparse then the savings can be dramatic.

5.6 More Trie Operations

The FIND, INSERT, and DELETE operations shown here can also be implemented via a hash table, as we will learn shortly. A properly constructed hash table has the same properties with respect to FIND, INSERT, and DELETE– all run in time proportional to the length of the key (since the hash function takes time proportional to the length of the key, and this time will dominate the operations in a good hash table), not the number of keys. Hash tables also are easy to modify so that they can deal with non-unique keys, which is extremely practical in some applications. Best of all, hash tables are relatively easy to implement and understand, and good implementations generally consume far less memory than tries. Given all of these properties, why would anyone still choose a trie?

- **Tries eliminate the need to choose a good hashing function.**

  If the hashing function is chosen poorly, or the data is pathologically bad, the degenerate behavior of hash tables is bad. There is no bad data set for a trie; all data sets are equally good, as far as time-complexity is concerned.

  On the other hand, some data sets may soak up more memory than others (although even in the worst case, memory requirements are bounded by an amount proportional to the total length of the keys).

- **A trie can preserve (and represent) an implicit ordering of the keys, based on an ordering of the characters of the alphabet.**

  This means that a trie can be used to sort a set of keys in much the same was a binary tree can– but a trie does not exhibit the degenerate $O(n^2)$ worst case behavior that results from inserting keys into a binary tree in a "bad" order. This ordering also means that it is possible to easily find the lexical predecessor or successor of any key, as well as several other properties of a key (see exercise 2 for an example).

- **Other information can be coded into the trie along with the keys.**

  Finally, there are algorithms that encode information related to the keys in the tree itself. For example, information about the prefix of every key is implicit in the trie, so the question of whether there exist any keys that begin with a particular substring is easy to answer. This sort of functionality is often used for command or filename completion. It is also absolutely indispensable to many adaptive coding techniques; for example the Lempel-Ziv compression algorithm is very easy to implement with a trie.[3]

[3] The Lempel-Ziv algorithm would make a fascinating topic for a final paper.

It is possible to construct hashing functions that allow efficient "prefix checking", as we will see in a later chapter, but it is difficult to construct one that can out-perform a trie.

5.7 Exercises

1. Write an algorithm that prints out all of the keys stored in a trie. Not counting the action of printing each key, it should run in time $O(n)$, where n is the number of nodes in the trie. (Printing the keys will require time $O(L)$, where L is the total length of all of the keys.) An auxillary data structure can be used, but its size must not be greater than $O(l)$, where l is the length of the longest key in the trie.

2. Using a patricia, as described in this chapter, show that inserting a new key can cause at most one node to be uncollapsed, and that the deletion of one key can cause at most one node to be collapsed.

3. Write an algorithm that takes a key k and finds all the keys in a trie that differ from k in exactly one character position. The algorithm should run in time $O(l)$, where l is the length of the key, and be independent of the number of keys in the trie.

   There are, of course, only $O(l)$ such keys possible, so a time complexity of $O(l)$ should be the worst case. Your goal is to rule out as many possibilities as you can as early as possible.

4. What is the time complexity of using a trie to sort a set of keys? (Hint– first choose how to measure the the "size" of the problem.)

Chapter 6

I/O of Data Structures

6.1 An Introduction

At this point in the semester, we've seen many of the fundamental kinds of data structures (and will cover more, later in the semester). We've generally explored these data structures from the perspective of how they can be represented and manipulated in *memory* using C constructs. However, it is often desirable or necessary to store data structures in some other form, such as a file, where the methods of representation available are quite different. The general nature of the techniques is a special case of the general process of *serialization* (the process of converting a data structure into a sequential or "serial" sequence of bytes so that it can be sent over a network, printed, or stored on a "serial" device such as a CD-ROM or magnetic tape). This chapter does not focus on how serialized data structures can be efficiently accessed or manipulated in more general ways– for example, how to perform a FIND operation on a binary tree stored as a disk file. The only operations we will focus on are how to store and restore data structures.

Rather than try to give solutions for every possible situation and discuss every tradeoff between different approaches, this chapter attempts to illustrate several key themes in data structure serialization, and show how they can be used, either alone or together.

6.1.1 A Word of Advice

Currently, computer processing speeds are enormously faster than I/O speeds. A typical PC can perform one million arithmetic operations in less time than it can read a single byte of information from its disk. Much the same situation exists in networking (although the range of network speeds varies widely)– a typical PC can perform a considerable amount of computation in the time it requires to send an inquiry to a remote server and receive its response. This means that for many kinds of problems, if the answer can be computed by purely computational means, it is more efficient to recompute the answer every time it is needed than it would be store the answer in a file and consult the file (or to ask a remote server to compute the answer to the question and return the answer).

Therefore, before asking *how* a data structure can be represented in a file, it usually makes sense to ask *whether* it makes any sense to store the data structure at all! For the purpose of this chapter, we will assume that the answer to this question is always yes, but in practice this is an important question to consider.

There are many situations where the data or its structure is truly important, usually because it cannot be recomputed by any known means. There many other cases where the data is relatively easy to recompute, however, and the line between easy and hard to compute is changing constantly. Early computers used tables to compute trigonometric functions, but today the idea of storing such tables on disk rather than recomputing them as necessary seems unthinkable.

6.2 Arrays

The first case study that we'll investigate is how to write and read arrays. We will begin with fixed-size one-dimensional arrays of integers, and then generalize our methods until we are able to deal with arrays of unknown dimension, extent, and type.

6.2.1 Arrays of Fixed Size

For the one-dimensional case, the solution is very easy– in C, one-dimensional arrays are stored as a contiguous block of memory. The standard I/O library makes it easy to read and write such blocks (i.e. using `fread` and `fwrite`). The only complication can occur when the arrays are of unknown extent or type, but these situations are discussed in later sections.

The two-dimensional case is more interesting, however, and is worth considering for a moment. If we were always reading and writing two-dimensional arrays of the same size (for example, NUM_ROWS by NUM_COLS, where NUM_ROWS and NUM_COLS are fixed numbers that everyone has agreed to), then reading and writing arrays would be a simple matter, as illustrated in figure 6.1.

Figure 6.1: Code to read and write 2-dimensional arrays of fixed size.

```
int array2dRead (int array[NUM_ROWS][NUM_COLS], FILE *in)
{

        int rc;

        rc = fread (array, sizeof (int), NUM_ROWS * NUM_COLS, in);
        return (rc == (NUM_ROWS * NUM_COLS));
}

int array2dWrite (int array[NUM_ROWS][NUM_COLS], FILE *out)
{
        int rc;

        rc = fwrite (array, sizeof (int), NUM_ROWS * NUM_COLS, out);
        return (rc == (NUM_ROWS * NUM_COLS));
}
```

6.2.2 Arrays of Unknown Extent

For many purposes, arrays of a fixed size are sufficient (at least until someone changes NUM_ROWS or NUM_COLS). However, in many cases, it is impossible (or impractical) to know ahead of time what the extents of the array actually are. Therefore, something we need to store (in addition to just the data in the array) is the size of the array itself.

One way to do this is to store the number of rows and columns of the array as the first two integers in the file, followed by the data, row by row.

A sketch of an implementation using the `array2d_t` structure from the *Pointers and Malloc* handout is shown in figure 6.2. This code omits all of the error checking that is generally necessary when using I/O, but gives the basic algorithm.

The first two integers, which represent the number of rows and columns, form a header for this serialized array; they tell the reader the information that they need to know in order to read the array. More information can be added to the header, if necessary; for example, an integer representing the type of each data element (`char`, `int`, or any other C type) could be used if the type of the array was unknown ahead of time.

Similarly, information about the format of the data elements themselves (size and byte-order) can be stored in the header, so that programs on machines with differently sized integers or different binary representations for integers can read the data correctly.

6.2.3 Arrays of Unknown Dimension

The algorithms from the previous sections are jury-rigged to work with two-dimensional arrays. They can be extended to work with arrays of higher dimension, using similar techniques, but there is another problem yet to be solved– how can we deal with an array that has an unknown number of dimensions?

One problem that would immediately arise is how to represent such an object in C in the first place– all the arrays we have seen so far have had a static number of dimensions, known at compile-time, and this is generally the way C programs are written. It is possible to write code that does otherwise, but I will not pursue this topic in this chapter. Instead of focusing on the C tricks necessary to use such arrays in C, I will focus on the methods used to read and write these arrays.

If the number of dimensions is known to be small, then a fixed-size header can be used, similar to the header used in the previous section to handle arrays of unknown extent. This header can list the extent of each dimension, with a sentinel value of zero to indicate the highest dimension, if it was less than the limit.

This technique works well if the number of dimensions can be bounded. However, this

Figure 6.2: Code to read and write 2-dimensional arrays of unknown size. Note the complete lack of error checking.

```
array2d_t *array2dReadDynamic (FILE *in)
{
        int rows, cols, i;
        array2d_t *array;

        fread (&rows, sizeof (int), 1, in);
        fread (&cols, sizeof (int), 1, in);

        array = arrayCreate (rows, cols);

        for (i = 0; i < rows; i++)
                fread (array->elems [i], sizeof (int), cols, in);

        return (array);
}

void array2dWriteDynamic (FILE *out, array2d_t *array)
{
        int i;

        fwrite (&array->rows, sizeof (int), 1, out);
        fwrite (&array->cols, sizeof (int), 1, out);

        for (i = 0; i < array->rows; i++)
                fwrite (array->elems [i], sizeof (int), array->cols, out);

        return ;
}
```

is not always the case. In this case, the approach is to use a dynamic header– the list of dimensions and their extents is *itself* a dynamic 1-dimensional array, represented in the manner shown in the previous section. In this representation, the input file contains:

1. Any information about the byte-order, word-size, etc of the represented data.
2. An integer specifying the number of dimensions.
3. An array of integers representing the extent in each dimension.
4. The data itself.

6.2.4 Arrays of Unknown Type

We have made a pair of simplifying assumptions in our discussion up to this point: first, we have assumed that both the reader and the writer implicitly know what the type of the array is (i.e. `int`, `double`, etc), and second that both the reader and writer use the same representation for these types.

This is not always the case. For example, different machines use different byte orders or different numbers of bytes to represent multi-byte objects, and different machines may use different binary representations to represent bytes (including bytes of different sizes).[1]

In order to successfully read data stored in the native binary representation of the writer, the reader must know what that format is (and how a translation into its own native format can be performed). Alternatively, all writers (no matter what their native representation) can write in a canonical format that all readers (no matter what their native representation) can read.

If writers always write in their own native format (and the readers perform whatever translation is necessary), then a marker must be added to the data that indicates exactly what representation was used to write it, so that the reader can choose the appropriate reading routines. This marker must be chosen in such a way (and written in a canonical, universally agreed-upon manner) so that all readers can unambiguously determine the way that the file was written.

[1] This is less and less of a worry each year; currently nearly every CPU uses 8-bit bytes, two's complement integers, and IEEE floating point.

6.2.5 Data Definition Languages

We began this section with an example where the reader and the writer implicitly agreed on the dimension, size, and type of the arrays, as well as the underlying representation of that type. By the final example in the previous section, all of the implicit assumptions had been removed, except one– the writer and reader have implicitly agreed that what is being written and read are arrays.

In some applications, even this assumption is invalid– the reader may have *no* idea what kind of thing the writer has left in the file, and may have to learn the type and format of the data from the data itself. This problem can be solved by adding still more information to the header, so that the data includes a description of its format, explaining how it should be interpreted by any readers. This description, however, must be in a form that is implicitly shared by the readers and writers– if the writer doesn't describe the data in a way that the reader can understand, then the data is useless to the reader.

6.3 Strings and String Fields

For this case study, we'll investigate how we can read and write records containing information represented by strings. In a general sense, strings are simply one-dimensional arrays, and can be handled as such, using the methods that we have already seen. Typically, however, there is useful information (such as information about the representation of the string) encoded in the string itself, and therefore we will consider strings as a separate case.

6.3.1 Delimited Strings

Imagine that we have a set of records that we wish to store in a file. Each record consists of a fixed number of fields, and each field consists of a string. Since some (or all) of the fields are of varying length, however, each record itself is represented by a string. How can we tell where each record and each field in the record begins and ends?

There are a number of approaches to solving this problem. The most common one, at least in the UNIX world, is to have each line in the file represent a single record, and have the boundaries between each field in the record denoted by a special string (usually just a single character like ":") called a *field delimiter* (or just a *delimiter*). For example, here's what a small file that represents a phone book might look like:

```
Ellard:Dan:496-6246
Ellard:Penny:643-9644
```

Writing records to a file in this format is simple; for each record, just do something along the lines of:

```
fprintf (fout, "%s:%s:%s\n", lastname, firstname, telephone);
```

Figure 6.3 is an example of an implementation of some code to read in a file of this form. You should note the following things about this code:

- It allows "comments" to be intermixed with the data. A comment line is any line that has a '`#`' as its first character.
- The logic for figuring out where each field begins and ends could be done with a single call to `sscanf`, and in fact the entire process of reading in each line and chopping it up into fields could be accomplished with a single call to `fscanf`. It might be interesting for you to figure out what the format string you would need to pass in to `fscanf` would be in order to accomplish this. Alternatively, finding the start and end of each field individually can be accomplished via repeated calls to `scanf` (or `fscanf`).

Given that I *could* have used `fscanf` (or some member of the `scanf` family, or `strtok`), it may seem odd that I did not. (In general, doing things the easy way is always my choice.) The reason is that these functions, while handy for many simple purposes, are hard to extend. In the next section, we'll extend these ideas significantly.

6.3.2 Escaped Strings

In the code in figure 6.3, there is no way to represent a field that contains the delimiting character itself. For example, if you ever try to enter a last name that has a ':' in it, our phone book routines will fail. In some applications, this is not a problem; if there exists a delimiter character that can be chosen that will not, due to some property of the data, ever appear in the data, then this approach is sufficient.

In the more general case, however, it might not be possible to choose a delimiter character at all, or at the very least it may not be possible to choose one that will work for all applications. Therefore, in this section we will discuss how delimited strings can be implemented in a manner which allows all of the characters in the character set to appear in any field.

The usual way to do this is to *escape* the field delimiting character by adding a special character or string (the *escape character*) before each character that needs to be "protected". You have already seen this technique in practice; `printf` uses the % character to escape the meaning of other characters.

Figure 6.3: Code to split a string into fields.

```
char **stringSplit (char *line, int field_count, char delim)
{
        char *eol, *start, *end;
        char **fields;
        int i;

        eol = strchr (line, '\n');
        if (eol != NULL)
                *eol = '\0';

        fields = (char **) malloc (field_count * sizeof (char *));

        for (i = 0; i < field_count; i++)
                fields [i] = NULL;

        start = line;
        for (i = 0; i < field_count; i++) {
                end = strchr (start, delim);
                if ((i == field_count - 1) && (end == NULL))
                        end = strchr (start, '\0');
                assert (end != NULL);

                fields [i] = malloc (sizeof (char) * (1 + end - start));

                strncpy (fields [i], start, end - start);
                fields [i][end - start] = '\0';

                if (*end != '\0')
                        start = end + 1;
                else
                        start = end;
        }
        assert (*start == '\0');
        return (fields);
}
```

Unfortunately, often it is unacceptable to use exactly the same scheme as `printf` (which uses % as the escape character, and escapes % itself with %%), since we may well want to allow two adjacent instances of the delimiter character to signify an empty field. For example, we might want "::" to signify an empty field, rather than a single ":" (but of course, we still want a way to represent a single ":"). What can be done is to use *another* character or string as the escape character, instead of using the delimiter as the escape.

The character that is most typically used in UNIX is backslash (\). So, if we ever want to represent the name of some eccentric named ”`foo:bar`” in our system, his name could be stored as ”`foo\:bar`”.

Of course, this immediately leads to another problem (that might not be immediately obvious). What if we then have to deal with another eccentric whose last name ends in ”\”? The answer, again, is that we need to escape the escape character itself! Luckily, we *can* use the escape character to escape itself (unlike the delimiter character, which cannot escape itself), so there the saga ends.

In our example code, we actually have *two* delimiting characters: the ":", and the newline. If we want to allow the user to embed newlines in their strings, then we can still accomplish this with the same method of escaping the newline- but we'd need to modify the code somewhat, because it uses `fgets` everywhere, and `fgets` treats newlines specially. However, despite the fact that it would require a little more coding to get working, algorithmically we now know exactly what we would need to do.

6.3.3 Annotated Strings

In many cases, it may be necessary to escape sequences of multiple characters. These substrings are *annotations* of the text. (In different contexts, different terms are used to describe this concept, but I'm going to use this single term to try to be consistent).

For example, in HTML the formatting commands are embedded directly in the text of the web pages, as annotations. In HTML, annotations generally begin with a ”<” and end with a ”>”. In C, you can think of comments as annotations that begin with `/*` and end with `*/` (although of course this fails to consider what happens when these sequences occur inside C strings). In these cases, the start and ending delimiters of annotations are different, but in some applications the same character is used to delimit both the start and end of an annotation (as double quotes are used to represent both the start and end of strings, for example).

Annotated strings are slightly more complicated to deal with than escaped strings, because there is (in the general case) no way to know ahead of time how long each annotation is. However, a state machine can be used to process the string and keep track of what sections represent normal text and what sections represent any annotations.

Note that annotated strings are often escaped as well: otherwise, there would be no way to represent the annotation string itself in the text. For example, in HTML, the escape character is "&".

Note that although HTML is a convenient (and ubiquitous) example of annotated text, it is also not a very good example, and certainly not a clean design. Instead of simply escaping the delimiters, each has its own special escape (for example, the string "<" is the escaped form of "<"). In addition, despite the fact that the escape character can be used to escape itself, the HTML escape character is escaped differently, as "&".

6.4 Binary Trees

The next problem we will tackle is how to store binary trees in a file. As always, there are a vast number of different situations that might arise, but luckily in the case of binary trees, these can divided into two general cases- binary search trees, and unordered binary trees.

6.4.1 Binary Search Trees

Imagine that we have an inorder binary tree consisting of integers- each node contains an integer value; the left subtree (if any) contains only keys less than the key of the node, and the right subtree (if any) contains keys that are greater. Imagine that we want to store this binary tree in a file in such a way that we can reconstruct it *exactly* as it was. In particular, our goal is to store the keys in such an order that when if we start with an empty tree, and we read them in again in the order they appear in the file and insert them into the tree, the resulting tree has exactly the same form as the original.

This problem may seem somewhat contrived, given the binary search trees that we have seen– the key properties of binary search trees (at least as far as we have discussed them so far) is that the keys are inorder, and (in order to minimize search time) the tree is reasonably balanced. Both of these properties can be attained by constructing the tree in a proper manner, regardless of the order that the keys are inserted into the tree. However, there are other kinds of binary search trees where there are other relationships between the nodes, and the exact shape of the tree actually has a great deal importance.[2]

[2] For example, multi-key search trees, where each level of the tree uses a different key.

Choosing a Traversal

Your first thought might be to write a routine that walks through the tree using one of the traversals that we've seen in class (inorder or depth-first, preorder, postorder, or level-order), writing out the value of each node as you encounter it- and you'd be right, but only if you picked the correct traversal!

If we use an inorder traversal, we'll put the values onto the disk in a sorted order- and as we've seen, if we build a binary tree from sorted keys (assuming that no balancing operations are performed), then we will always get a tree that looks like a linked list. Unless by some incredible coincidence all the trees we want to store happen to look like this, the resulting trees will not be identical to the originals.

If we use a postorder traversal, it won't work either (although at least the trees won't always come out looking like linked lists). If we try a few different examples, we can see see that it just doesn't do what we need, but after some fooling around you might start to catch on to *why* neither inorder or postorder does the right thing- which leads us to a preorder or level-order traversal.

Recall that when we build a binary search tree using the standard insertion routines, we build from the top down– nodes are always added as leaves. Therefore, in order to reconstruct the original tree, the following property must hold- *before a key is inserted into the tree, the key which was the key of its parent node in the original tree must already be in the tree.* Preorder or level-order traversal will do what we need, since each always visits parents before their children, and so parents will always appear before their children in the file.

Since preorder is trivial to code and efficient (while level-order slightly more complex and may require more space), it makes sense to use preorder. It's interesting to note that there are a number of other traversals that share this property.[3] However, since preorder is the easiest to code and understand, there is no reason not to use it.

For an example of an implementation, see figure 6.4.

This code is very simple– partly because it doesn't do much error checking or attempt to recover from any errors, but mostly because the algorithm is very short and simple. Note that the `treeInsert` function can be any ordinary tree insertion routine, but it must not be one that attempts to balance or otherwise modify the existing shape of the tree in any way other than creating a new leaf!

The code in figure 6.4 only reads and writes integer keys. It could be extended to work with binary search trees that use some other kind of key, and have values composed of

[3] In fact, preorder isn't the first one that I thought of when I first had to solve this problem myself, although I wish it had been, because it would have saved me a lot of work!

Figure 6.4: Routines to read and write binary search trees with integer keys.

```
void treeWrite (tree_t *tree, FILE *fout)
{
	int rc;

	if (tree == NULL)
		return ;

	/* Write the key (and value, if any). */
	rc = fwrite (&tree->key, sizeof (int), 1, fout);
	assert (rc == 1);

	treeWrite (tree->left, fout);
	treeWrite (tree->right, fout);
	return ;
}

tree_t *treeRead (FILE *fin)
{
	tree_t *tree = NULL;
	int key, rc;

	for (;;) {
		/* Read the key (and value, if any). */
		rc = fread (&key, sizeof (int), 1, fin);
		if (rc != 1)
			break;

		tree = treeInsert (tree, key);
	}
	return (tree);
}
```

arbitrary records of strings or other types, using the same sort of methods that we used in the previous section that described how to deal with strings.

6.4.2 Unordered Binary Trees

Now that we've conquered binary search trees, let's tackle something a little more complicated- "unordered" binary trees, such as expression trees or animals trees (like those that we used in the `animals` program).

The crucial difference between this type of tree and a binary search tree is that there isn't any general way to compare any two arbitrary keys in order to determine which one should come before the other. We know at a glance whether one integer is larger or smaller than another in a binary search tree with integer keys, but in an animals tree there's no way to tell at a glance whether "cat" will appear to the left or right of "dog" or "bear"- we must traverse the tree, answering questions about these beasts in order to know where they will be located.

So, how can we proceed? There are several possibilities.

An Explicit Reconstruction

One approach is to encode the explicit sequence of operations that was used to construct the tree itself.

For an animals tree, this reduces to storing an explicit sequence of "yes", "no", questions, and animal names that can be used to reconstruct the original tree. This does *not* mean that the original game itself needs to be saved: only the information represented by the tree. In effect, we would be recording an abbreviated game of animals, in such a way that reading it back in would simulate replaying the game. This requires storing the relationship between each node and all of its ancestors (i.e. the sequence of "yes" and "no"s that lead to each node).

Like in the previous case, we would need to do a preorder walk of the tree when writing it, in order to ensure that the ancestors of each node are installed before we try to install the node. However, depending on how the tree is actually constructed, it is not always the case that the ancestor questions of an animal node preceeded the animal in the game– in fact, the opposite is the case. Therefore, some care is needed.

For animals, this seems like a complex and inefficient task, but for other problems, this is not necessarily a bad approach. If constructing the tree is not terribly inefficient, or the contents of the tree can be represented easily in some other manner, then this approach works well. For example, in an expression tree, it is very efficient to convert an infix expression into an expression tree (or vice versa)– and the infix expression is a more compact representation of the expression (which makes it more efficient to store).

An Implicit Representation

A better approach (in most cases) is to record the structure of the tree implicitly instead of explicitly recording all of the information used to construct the tree. This requires less storage and is somewhat more flexible.

For example, to write out an animals tree, we do a preorder walk, but along with the contents of each node we record whether the node is a question node or an animal node. When we read the tree back in, we read it again in preorder, and know where we are in the recursion based on whether the node is a question node (in which case we need to recursively read in the "yes" and "no" children) or an animal (the base case of the recursion).

An example of this algorithm is given in C in figure 6.5. The reading algorithm, though also quite simple, is quite a bit less obvious than the algorithm for reading a binary search tree, and you may need to try it once or twice to convince yourself that it actually works.

Some particular details to note about the code in figure 6.5:

- This code assumes that the tree is exactly like the trees we used in `animals`- each node is either an internal node with two children, or a leaf with zero children. This code could be extended to handle trees where nodes can have a single left or right child.

  This can be done by modifying the `aniTreeWrite` function to denote "question" nodes with a single left child with `QUESTION_LEFT_CHAR`, and nodes with a single right child with `QUESTION_RIGHT_CHAR`, and then modify the case statement in `aniTreeRead` as shown in figure 6.6.

- When the original call to `aniTreeRead` returns, it is assumed that the entire tree has been read in. Real programs might do some error checking to make sure that there isn't any more data in the file, and indicate some sort of error. (In general, this code is almost entirely lacking in error checking.)

Animal trees have a relatively simple structure, so these routines are quite simple. In the more general case, binary trees can be represented by strings using the method sketched in algorithm 6.1:

An example of a tree represented this way is shown in figure 6.7. Note that this method can easily be extended to work for M-ary trees as well.

Figure 6.5: Routines to read and write an Animals tree.

```
#define ANIMAL_NODE_CHAR        'A'
#define QUESTION_NODE_CHAR      'Q'

void aniTreeWrite (tree_t *tree, FILE *fout)
{
        if (tree == NULL)
                return ;
        else if ((tree->yes == NULL) && (tree->no == NULL)) {
                fprintf (fout, "%c%s\n", ANIMAL_NODE_CHAR, tree->animal);
        }
        else {
                fprintf (fout, "%c%s\n", QUESTION_NODE_CHAR, tree->question);
                aniTreeWrite (tree->yes, fout);
                aniTreeWrite (tree->no, fout);
        }
}

tree_t *aniTreeRead (FILE *fin)
{
        tree_t *node = aniNodeCreate (NULL, NULL, NULL, NULL);
        char type = getChar (fin);      /* get the type char.   */
        char *text = getLine (fin);     /* read a line...       */

        switch (type) {
                case ANIMAL_NODE_CHAR   :
                        node->animal = text;
                        break;
                case QUESTION_NODE_CHAR :
                        node->question = text;
                        node->yes = aniTreeRead (fin);
                        node->no = aniTreeRead (fin);
                        break;
        }
        return (node);
}
```

Figure 6.6: Dealing with question nodes that have only a left or right child.

```
case QUESTION_LEFT_CHAR : {
        node->question = text;
        node->yes = aniTreeRead (fin);
        node->no  = NULL;
        break;
}
case QUESTION_RIGHT_CHAR: {
        node->question = text;
        node->yes = NULL;
        node->no  = aniTreeRead (fin);
        break;
}
```

Algorithm 6.1 (Representing a Binary Tree as a String) *An algorithm to generate a string that represents a binary tree.*

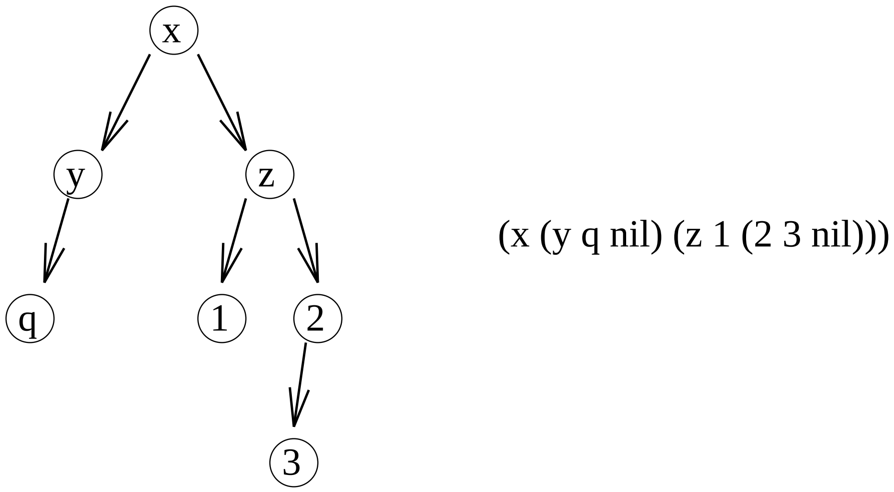
Beginning with an empty string, perform a preorder traversal of the tree. At each node, add the representation of that node to the end of the string, according to the following rules:

- If the node is a leaf, then it represented by the string representation of its key (and value, if any). Any instances of any of the delimeters used by the tree representation must be escaped if they appear in this string.

- If the node is NULL, it is represented by a special annotation. In this example, the string `nil` will be used. (Note that any instances of the string `nil` must be escaped.)

- An internal node is represented by a left node delimeter (in this example, a left parenthesis), followed by the string represenation of its key (and value, if any), a field delimeter (in this example, a space), the string representation of its left descendants, a field delimeter, the string representation of its right descendants, and finally a right delimeter (in this example, a right parenthesis). Note that any instances of the left or right node delimeters or the field delimeter in the key or value must be escaped.

Figure 6.7: String representation of a binary tree.



Adding Inorder Keys

The simplest (and perhaps most flexible) approach is to add an extra integer "position" field to each node in the tree. This field is used *only* by the I/O routines. This field is filled in by an inorder traversal of the tree immediately before it is written, and then used as the key to reconstruct the tree.

The algorithms for reading and writing are given in algorithms 6.2 and 6.3.

Algorithm 6.2 (Unordered Binary Tree Write) *Algorithm to write an unordered binary tree to file.*

1. Do an inorder traversal of the entire tree, filling in the position field with the inorder position of each node in the tree.

2. Write the tree as an inorder tree, using this field as the key.

This approach is simple and fast, but requires some storage overhead. The fact that it requires storing information in the tree itself means that this approach can only be used when the tree can be modified (and fields for this purpose were set aside when the data

Algorithm 6.3 (Unordered Binary Tree Read) *Algorithm to read an unordered binary tree (written by algorithm 6.2) from file.*

Read the tree as a normal inorder tree (using the algorithm described earlier), using the position field as the key for insertion.

---

structure was created). If this is not the case, then this algorithm cannot be used as stated here, although its central idea can be used. (See the exercises.)

6.5 Exercises

1. Prove the statement (from section 6.3.2) that it is always possible that the escape character can be used to escape itself (assuming that the delimiting character is different from the escape character).

2. Our method of reading a binary search tree is $O(n \log n)$, since it inserts n items into the tree, with an expected time of $O(\log n)$ per insertion. (In the worst case, of course, each insertion may be $O(n)$, so that the reconstruction requires time $O(n^2)$). This is not usually a matter of concern, since the $O(n)$ disk operations necessary will be so much slower than the tree insertion. For very large and unbalanced trees, however, this could present a problem.

   Show how it is possible, by storing an additional $O(n)$ items of data with the tree (and using $O(n)$ additional memory while the tree is being reconstructed) to *always* reconstruct the tree in time $O(n)$. (If you find a way achieve time $O(n)$ with less then $O(n)$ additional storage, feel free to explain how this can be accomplished– but be careful!)

   Hint– consider the techniques for storing unordered binary trees.

3. Show how it is possible to use the idea behind algorithms 6.2 and 6.3 so that unordered trees can be written and read using the basic ideas behind these algorithms but *without* modifying the original tree or marking any of its nodes in any way. $O(n)$ extra storage space can be used (both in the file and in memory).

Chapter 7

The Monte Carlo Method

The "Monte Carlo method" is a general probabilistic algorithm for estimating the probability of an event based on the a number of random trials.

7.1 Randomized Algorithms and Probability

A *randomized algorithm* is an algorithm that makes at least one decision based upon a random choice, rather than a fixed rule.

At first glance, it may seem surprising (or even unbelievable) that incorporating an element of chance into an algorithm can result in a correct and efficient algorithm, but as we will see in this chapter, there are many situations where randomness is extremely effective.

Randomized algorithms generally depend upon the laws of probability, and therefore before we begin studying randomized algorithms, it makes sense to formally define some of the terms and concepts of probability.

- An **experiment** is any procedure that can be repeated any number of times, and has a well-defined set of outcomes. (In order to be interesting from a statistical perspective, the result of the procedure is generally not known in advance.)

- An **sample outcome** (often denoted s) is a member of the set of outcomes of an experiment.

- The **sample space** (often denoted S) is the complete set of all possible sample outcomes of an experiment.

- An **event** is any set of sample outcomes. When the outcome of an experiment is a member of this set, the event is said to **occur**.
- Two events are **independent** if the occurance of one of the events is not related to the occurance of the other. For example, if we flip a fair coin twice, then the event "first flip is heads" is independant of the event "second flip is heads". However, the events "first flip is heads" and "first flip is tails" are *not* independent, because the occurance of one of these events depends on whether the other has occured. In this case, the correspondance is absolute– but this is not always the case.
- A **trial** is the actual "execution" of an experiment. For example, the procedure of tossing a coin and observing whether the result is heads or tails is an experiment, but the act of actually flipping the coin is a trial.
- A **population** is the set of all unique trials. This set may be infinite. For example, the population of all coin tosses is infinite, because we can perform as many unique trials as we like by flipping the coin repeatedly. In other situations, the population may be finite– for example, if our experiment is to ask a person enrolled in S-Q whether their last name contains the letter 'e', then the set of unique trials is the set consisting of one trial for each person enrolled in the course.
- A **sample** is a set of trials. A **random sample** is a set of trials selected at random from the population.

For example, consider the act of rolling a 6-sided die. The procedure consisting of rolling the die and reporting which face is visible is the experiment; and each time we actually perform this procedure we are performing a trial. We can roll the die as often as we like, and every time we roll the die the outcome (i.e. which side of the die is showing) must be one of the 6 sides.[1] Each possible side of the die is a possible sample outcome, and these six possibilities constitute the entire sample space.

We can define a number of possible events for this experiment– for example, rolling any particular number, or rolling an odd number, or rolling a number less than four.

[1] We assume that the die must always come to rest on one of its faces, not balanced on an edge or a corner.

7.2 Monte Carlo Simulation

Monte Carlo simulations are one of the most common randomized algorithms. A Monte Carlo simulation is a method of estimating the value of an unknown quantity by making use of the principles of inferential statistics.

In brief (since this is a course in algorithms, not probability and statistics), the guiding principle of inferential statistics is that a random sample tends to exhibit the same properties as the population from which it is drawn.

For example, if you flip a coin 10 times, and it comes up heads every single time, then you might start to form an opinion that the coin is not fair, and that it is biased towards coming up heads– you might begin to believe that the population of all possible flips of this coin contained more heads than tails.[2] If you flip the same coin 100 more times, and it still comes up heads every single time, it would probably be hard to convince you that the coin was fair at all (in fact, you might start to believe that it had two heads!).

On the other hand, if you flipped the coin 100 times and roughly half of the time it came up heads, then you would probably believe that it was a fair coin– or at least not unfair to any obvious degree.

In both cases, your belief whether the coin is fair or not is based on the intuition that the behavior of a sample of 100 flips is similar to the behavior of the population of all flips of your coin. Luckily, this belief is not without basis in fact– and we can make use of this fact to achieve good estimates of probabilities with relatively little effort. This method is formalized in algorithm 7.1, which shows a method of estimating p for a coin, based on a set of random trials.

7.2.1 Accuracy of a Monte Carlo Simulation

When we use algorithm 7.1 to perform a Monte Carlo simulation, we are trusting that fate will not give us an unusual or freakish sample– we are trusting that the true probability is reflected in our observations. In the long run, we can generally rely upon this assumption– although it is important to know how many samples we will have to take before this assumption holds.

While we trust in luck to ensure that our sample is not wildly unusual, we must also realize that it is extremely unlikely that fate will provide a sample that is *exactly* like the

[2] From a mathematical perspective, it is incorrect to so blithely make a comparison of the cardinality of these two infinite sets. Although most people's intuition about the nature of infinity is far from correct, most people do find probability reasonably intuitive, and I appeal to that intuition here instead of embarking on a lengthy mathematical digression.

Algorithm 7.1 (Monte Carlo Simulation) *An algorithm to estimate the probability p that a coin will come up heads.*

- Flip the coin N times. Count the number of times heads appears, and call this number M.
- As N grows larger and larger, we expect that, through the graces of probability, the ratio M/N will become very close to the actual ratio p.

  Thus we estimate p as (M/N).

---

population.[3] Consider the example of flipping a fair coin 1000 times. While it is likely that the number of heads and tails will be close to 500, it is also unlikely that they will both be *exactly* 500. What is far more likely is that the number of heads will fall in some range around 500.

This leaves us with two questions to ponder:

1. How many samples do we need to take in order to safely assume that we have taken "enough" samples?
2. How accurate do we believe that our estimate is, given a certain number of samples?

As you might guess, the answers to these two questions are related– the more sure we want to be that our sample is good, the more samples we need to take, and the more samples we take, the more likely it is that we will get a more accurate estimate.

Unfortunately, it is never possible to achieve perfect accuracy through sampling unless you sample the entire population– no matter how many samples you take, we can never be sure that the sample set is typical until we verify every last element of the population (and since we are usually dealing with infinite populations, this is simply impossible). Of course, this is not to say that our estimates are not perfectly correct– we might arrive at the exactly correct value, but the trouble is that *we can't know that our estimate is correct*! All we can measure is how *likely* it is to be correct.

[3]In some cases, such as we will explore later in the chapter, this is not even *possible*, no matter how kindly fate smiles upon our program– our estimates will always be rational numbers, and not all probabilities are rational.

Since this is a course in data structures and algorithms, and not a course in statistics, rather than deriving the relationship between the sample size and accuracy estimate from first principles (which requires the first week or two of a course in mathematical statistics to accomplish), it is now time to begin making gross (but useful) generalizations.

The relationship between the sample size and our estimate of the accuracy of an estimate arrived at from a sample is given by the following equations:

$$e = \frac{z}{2\sqrt{n}} \tag{7.1}$$

$$n = \left(\frac{z}{2e}\right)^2 \tag{7.2}$$

In these equations, n is the sample size, and e is the "margin of error", which is the fraction by which we believe our estimate may be incorrect. The variable z is a function of how *confident* we desire to be in our estimate of e– the probability that our estimate is within e of the true value. Computing z is a laborious process (and it was even more laborious before computers were given this task). However, the following table contains values of z for some common confidence levels:

z	Confidence Level
1.96	95%
2.58	99%
3.29	99.9%

A few very important caveats about these equations:

- They should not be used with small n. The assumptions upon which they are based break down when n is less than 30.
- Similarly, they should not be used with probabilities that are extremely near zero or one unless a large number of samples are drawn. One rule of thumb is that the estimate should be based on at least 5 trials with both outcomes– so if you are estimating the probability of an event that has a very low true probability, you may have to take a large number of samples before you have any evidence at all that the probability is non-zero– but if you happen to draw a positive sample in one of the first trials and stop soon thereafter, your probability estimate may be wildly high.

- These equations are pessimistic. Assumming that the previous two conditions are met, they generally give margins of errors that are too wide (or suggest that you should perform more trials than you really need to). Personally, this is the direction I prefer to err in– I would rather believe that my estimate is less accurate than it is, instead of thinking that it is more accurate than the facts would support. However, if you are trying to perform the absolute minimum number of trials necessary to achieve a given level of confidence, you may wish to find a tighter bound.

Computing the Margin of Error

Returning to our example of flipping a coin 1000 times, if the coin comes up heads exactly 500 times, our estimate of the probability of the coin coming up heads is 0.5. However, now we can also compute our "margin of error" for this estimate. With 95% confidence, we can say that our estimate is accurate to within 3.1%:

$$e = \frac{z}{2\sqrt{n}} = \frac{1.96}{2\sqrt{1000}} = 0.031$$

Similarly, we can compute our "margin of error" with a 99% level of confidence, using the same method, but with a z of 2.58:

$$e = \frac{z}{2\sqrt{n}} = \frac{2.58}{2\sqrt{1000}} = 0.041$$

Notice that our margin of error has increased. In order to gain more confidence in our estimate, we have had to widen the margin. Note that the *only* way to achieve 100% confidence is to widen the margin of error to be so wide that it spans the entire range from 0.0 to 1.0– the only way that we can be 100% confident that our margin of error contains the correct probability is to include all probabilities!

Returning to the coin example, if our estimated p is 0.5, and our margin of error (based on 1000 trials, at a 95% level of confidence) is 0.031, then we can restate our estimate as 0.5 ± 0.031 (with a 95% level of confidence). This way of expressing the estimate together with the margin of error is called a *confidence interval*.

In this case, based on our 1000 trials, we can say that we are 95% confident that the true probability of the coin coming up heads is within the range from 0.469 to 0.531. The true probability *might* be outside this range, but we are 95% confident that it is not.

Note that if our margin of error is large enough (either because we are requiring a very high confidence level, or the number of samples is too small), the confidence interval may include values outside the range from 0 to 1. These values should be discarded, because they are not valid probabilities.

Computing the Number of Trials

In the last section, we computed the confidence interval for a sample of 1000 trials. In this section, we will ask the problem from the other perspective– in order to get an estimate of the probability p of a coin (i.e. the probability that the coin will come up heads) that is accurate to within 5%, with a 95% level of confidence, how many trials do I need to perform?

Returning the equation:

$$n = \left(\frac{z}{2e}\right)^2 = \left(\frac{1.96}{2(0.05)}\right)^2 = \left(\frac{1.96}{0.1)}\right)^2 = (19.6)^2 = 384$$

Therefore, we can be 95% confident that an estimate of p based on 384 coin flips will be accurate to within 5% of the true p. (If we want to be more confident, or to achieve a more accurate estimate with the same level of confidence, then more coin flips will be necessary.)

7.3 Monte Carlo Integration

One good use of the Monte Carlo method is called Monte Carlo integration. The goal of integration is to find the area of a shape X; Monte Carlo achieves an estimate of the area by guessing. The method is almost entirely the same as we saw earlier, except that instead of estimating the p of a coin, we are estimating an area. The trick is to reduce this problem to the problem of estimating p of a coin. Our coin flips are performed by randomly choosing points from an area Y that completely encloses X, and whose area we know– if the point is inside X, then we say that it is "heads", but if it is outside X, then it is "tails". Once we have an estimate on the p for this imaginary coin, we can multiply p by the area of Y to get an estimate of the area of X.

This is described in more detail in algorithm 7.2.

It is important to note that if we cannot find a region Y or a function **inside**(p) that meets the criteria, then this algorithm cannot be used. Metaphorically, this is because we have no way of determining whether the "coin" we just flipped came up heads or tails!

As a concrete example, consider the problem of estimating the area of the quarter circle with a center at zero and a radius of 1. If you recall geometry, you will remember that the area of any circle is πr^2, where r is the radius of the circle, Thus, a complete circle with a radius of 1 will have an area of π, so our quarter circle will have an area of $\pi/4$.

However, if you had forgotten geometry (or simply forgotten what the value of π is), then finding the area of this quarter circle would represent a difficult problem to solve without calculus. Let us imagine, for the moment, that this is the case.

Algorithm 7.2 (Monte Carlo Integration) *An algorithm to find the area of a shape* X.

Let **inside**(x, y) be a function that determines whether point (x, y) is inside shape X or not.

- Find a region Y that completely encloses X, and whose area is known. (For example, Y can be any rectangle that completely surrounds X.)
- Randomly choose N points inside Y, and for each point $t = (x, y)$ use **inside**(t) to determine whether the point p is also inside X, or whether it is inside Y but *not* inside X. Let M be the number of points that fall inside X.
- As N grows larger and larger, we expect that, through the graces of probability, the ratio M/N will become very close to the actual ratio of the area of X to the area of Y.

  Thus we estimate the area of X as (M/N) multiplied by the area of Y (which we already know).

If Y is rectangular, then code to implement this algorithm is shown in figure 7.1.

By consulting a mathematics textbook and looking at the figure, however, you discover the following pieces of useful information:

1. The quarter circle falls entirely in the square with corners (0,0), (0,1), (1,1), and (1,0).

   This defines Y for our purposes.

2. A circle centered at the origin and radius r is defined as the set of points that satisfy the equation $x^2 + y^2 = r^2$. Therefore, since r in our case is 1, then any point such that $x^2 + y^2 \leq 1$ is inside the circle, and all other points are outside it.

   Therefore, for our quarter circle, if $0 \leq x, y \leq 1$, then:

$$\textbf{inside}(x, y) = \begin{cases} \textbf{TRUE if } x^2 + y^2 \leq 1 \\ \\ \textbf{FALSE otherwise}. \end{cases}$$

   This defines the `inside` function for our quarter circle.

This leads directly to the implementation of the `inside` function shown in figure 7.2.

Figure 7.1: Code to perform Monte Carlo integration.

```
double  findArea (int N, double xMin, double xMax,
                double yMin, double yMax)
{
        double x, y;
        double area = (xMax - xMin) * (yMax - yMin);
        int M = 0;
        int i;

        for (i = 0; i < N; i++) {
                x = randomDouble (xMin, xMax);
                y = randomDouble (yMin, yMax);

                if (inside (x, y))
                        M++;
        }

        return ((area * M) / N);
}
```

Figure 7.2: The `inside` function for estimating π, using a circle with radius 1, centered at the origin.

```
int inside (double x, double y)
{

        return ((((x * x) + (y * y)) < 1) ? 1 : 0);
}

```

7.4 Exercises

1. Imagine that you want to estimate the proportion of the population of American voters who consider themselves Democrats. Assuming that you can select randomly from among the voters, and each person you select tells us the truth (two conditions that are hard to satisfy in actual practice), and assuming that both Democrats and non-Democrats are not extremely rare:

   (a) How many voters do you need to ask in order to achieve an estimate that you are 95% confident is within 10% of the true proportion?

   (b) How many voters do you need to ask in order to achieve an estimate that you are 99% confident is within 2% of the true proportion?

Chapter 8

String Searching

8.1 The General Problem

The string searching problem can be defined as the following: given a string P and a string T, determine whether or not P appears as a substring anywhere in T. If so, return the offset of the start of the occurrence of P in T. If not, indicate that no such occurrence exists by returning -1.

Note that string searching algorithms can, in general, be extended to find *all* of the occurrence of P in T instead of only the first, but for the sake of simplicity we will initially only consider the problem of finding the *first* occurrence.

For example, let $P =$ "ring" and $T =$ "string searching". The string "ring" does appear in T, starting in position 2 (using C's zero-based indexing).

8.1.1 Terms and Definitions

- The **text** is the string that we are searching.
- The **pattern** is the string that we are searching for.
- A **shift** is an offset into a string.

In the previous example, the text is "string searching", the pattern is "ring", and the shift of the occurrence is 2.

An algorithm to solve the string searching problem immediately suggests itself– starting with a shift i of 0, simply compare the substring of T starting at i with the pattern P.

Continue testing until a match is found, or the end of T is reached. This algorithm is presented in C in algorithm 8.1.

Algorithm 8.1 (Brute-Force String Search) *Brute-Force string searching algorithm to determine whether there exists any occurrence of pattern P (of length M) in text T (of length N). If so, returns the index of the start of the occurrence; otherwise returns -1.*

```
long stringSearchBF (unsigned char *P, long M, unsigned char *T, long N)
{
        long i, j;

        for (i = 0; i <= N - M; i++) {
                for (j = 0; j < M; j++) {
                        if (P[j] != T[i + j]) {
                                break;
                        }
                }
                if (j == M) {
                        return (i);
                }
        }
        return (-1);
}
```

One important optimization that appears in this code is that the attempt to match P and each substring of T is abandoned as soon as any character in P fails to match the substring from T. Depending on the nature of P and T, this simple optimization can have a profound impact on the execution speed of the algorithm!

In the worst case, this algorithm performs $M(N-M)$ character comparisons (comparing each of the M characters in P to each of the characters in each of the $N-N$ possible substrings of T). If mismatches are recognized earlier, however, then substantially fewer comparisons are necessary. For example, if the first character of P does not appear anywhere in T, or only as part of a match, then the attempts to match each substring will all fail immediately, and this algorithm will only require $N-M$ character comparisons.

In the rest of this chapter, we will explore several ways to reduce the amount of work required in the worst case. We will begin with the Rabin-Karp algorithm, and then progress on to the Knuth-Morris-Pratt and Boyer-Moore algorithms.

It must be noted that for many purposes, the brute force string searching algorithm is sufficient. It is small and simple, and can be coded in an efficient manner that takes advantage of the underlying hardware. For short patterns, or even long patterns that begin with "rare" prefixes, the brute-force method is as fast as any.

8.2 The Rabin-Karp Algorithm

8.2.1 String Searching via Hashing

We can use the technique of hashing (explored earlier in the semester) as a way of testing whether two strings are the same– if two strings hash to the same value, then they *might* be the same string, but if they hash to different values then then are definitely *not* the same. This property can be used in a string searching algorithm as shown in algorithm 8.2.

Algorithm 8.2 (Hash Matching) *A string-searching algorithm that uses hashing to find probable matches.*

Given pattern S (of length M) and text T (of length N):

1. Compute $h_p = \textbf{hash}(p)$.
2. For each substring t_i (of length M) of T, compute $h_i = \textbf{hash}(t_i)$. If $h_i == h_p$, test whether $S == t$. If so, then return i.

However, at first analysis, this seems like a terrible idea: good hash functions need to look at every character in the string. If the length of the pattern is M, and the hash function looks at each character in the substring, then computing a hash of a substring of length M takes (at least) $O(M)$ to compute. If the hashing function looks at every character in the string, then this algorithm will may need to perform an $O(N)$ calculation at each of the $M - N$ possible positions in the text, giving an algorithm that is $O(M(M - N))$.

If you recall from section 8.1.1, $O(M(M - N))$ is the time complexity of the *worst-case* of the naive algorithm! This is certainly a bad sign– it would seem that by using hashing,

Figure 8.1: A hashing function to compute the hash of string $\alpha_0 \cdots \alpha_{n-1}$, where the size of the character set is b and p is relatively prime to b.

$$\mathbf{hash}_{b,p}(\alpha_0 \cdots \alpha_{n-1}) = \left(\sum_{i=0}^{n-1} \alpha_{n-i-1} b^i \right) \mathbf{mod}\ p$$

we're dooming ourselves to *always* do the amount of work that would otherwise be done by worst case of the brute-force algorithm.

As you might guess, however, there is a way around this problem. Although there is no general way to reduce the amount of work required to perform a *single* hash, there exist hashing functions that can be computed very efficiently for a *group* of strings that are related to each other. In this algorithm, each substring t_i is closely related to the next substring: the next substring simply has one character chopped off of one end and a new character added to the other end.

Can we devise a good hashing function that takes advantage of this fact? There are several, but one that is commonly used is shown in figure 8.1. This hashing function treats the string as a large number, where each character in the array represents a digit in a number in base b. The modulo of this number is then taken by p to give the hash value. In order to get good results, this p should be chosen so that it and b are relatively prime; this is easily accomplished by having b or p be prime.

Properties of Modulo

The hash function given in figure 8.1, as stated above, is clumsy (or impossible) to implement in C. The problem is that integers in C have a fixed number of bits (usually 32 or 64, though 16-bit machines are not unusual). Depending on the values of b and n, it may be inevitable that the $\sum \alpha_{n-i-1} b^i$ term will be too large to fit into an `int` (or *any* ordinary C type). For example, if $b = 256$, then n must be 4 or less in order for the result of the summation to fit into a 32-bit integer.

Luckily, we can exploit an important property of the modulo operation– taking the modulus of intermediate steps of an addition, multiplication, or subtraction calculation has no effect on the outcome of the calculation. For example, for any α and β:

$$(\alpha + \beta) \,\%\, m \quad = \quad ((\alpha \,\%\, m) + (\beta \,\%\, m)) \,\%\, m$$

$$\begin{aligned}
(\alpha\beta) \,\%\, m &= ((\alpha \,\%\, m)(\beta \,\%\, m)) \,\%\, m \\
(\alpha - \beta) \,\%\, m &= ((\alpha \,\%\, m) - (\beta \,\%\, m)) \,\%\, m
\end{aligned}$$

This means that by performing the modulo operation early and often, we can compute the modulus of the results of operations on very large numbers without actually ever worrying about the size of the numbers, as long as the modulus we are using is not very large.

For example, imagine that we want to compute $\alpha\beta \,\%\, b$, where α and β are very large numbers, but b is a small number. The product $\alpha\beta$ will be an extremely large number– but we need not ever deal with it, because $\alpha\beta \,\%\, b = (\alpha \,\%\, b)(\beta \,\%\, b) \,\%\, b$, and by definition, the result must always be somewhere in the range 0 and $b - 1$.

By aggressively taking the modulus during intermediate steps, we can avoid using very large numbers at all.

An Example

Imagine that we want to compute $2345_{10} \,\%\, 7$. It is useful to first compute several powers of 10, $\%$ 7:

$$\begin{aligned}
1 \,\%\, 7 &= 1 \\
10 \,\%\, 7 &= (10 \times 1) \,\%\, 7 \\
&= ((10 \,\%\, 7) \times 1) \,\%\, 7 \\
&= (3 \times 1) \,\%\, 7 \\
&= 3 \\
100 \,\%\, 7 &= (10 \times 10) \,\%\, 7 \\
&= ((10 \,\%\, 7) \times (10 \,\%\, 7)) \,\%\, 7 \\
&= (3 \times 3) \,\%\, 7 \\
&= 2 \\
1000 \,\%\, 7 &= (100 \times 10) \,\%\, 7 \\
&= ((100 \,\%\, 7) \times (10 \,\%\, 7)) \,\%\, 7 \\
&= (2 \times 3) \,\%\, 7 \\
&= 6 \\
10000 \,\%\, 7 &= (1000 \times 10) \,\%\, 7 \\
&= ((1000 \,\%\, 7) \times (10 \,\%\, 7)) \,\%\, 7
\end{aligned}$$

$$
\begin{aligned}
&= (6 \times 3) \ \% \ 7 \\
&= 4
\end{aligned}
$$

Given these values, it is easy to compute 2345 % 7:

$$
\begin{aligned}
2345 \ \% \ 7 &= (2000 + 300 + 40 + 5) \ \% \ 7 \\
&= ((2 \times 1000 \ \% \ 7) + (3 \times 100 \ \% \ 7) + (4 \times 10 \ \% \ 7) + 5) \ \% \ 7 \\
&= ((2 \times 6) \ \% \ 7 + (3 \times 2) \ \% \ 7 + (4 \times 3) \ \% \ 7 + 5) \ \% \ 7 \\
&= (5 + 6 + 5 + 5) \ \% \ 7 \\
&= 21 \ \% \ 7 \\
&= 0
\end{aligned}
$$

8.2.2 The Trick That Makes Rabin-Karp Fast

The trick that makes the Rabin-Karp string searching algorithm efficient is the fact that we have chosen a hash function which is easy to compute for adjacent substrings– once we know what the hash value of the first substring is, we can compute the hash value of the next in time $O(1)$. This is accomplished by taking advantage, once again, of the properties of the modulo operation.

For example, imagine we are that the text and pattern string are drawn from the alphabet consisting only the characters 0 through 9. The text is "234591", and we are searching for a pattern of length 4. We have already computed 2345 % 7. How can we use this to help us compute 3459 % 7?

Observe that 2345 can be transformed into 3459 by the following steps:

1. Multiply 2345 by 10, producing 23450.

2. Subtract 20000 from 23450, leaving 3450. This erases the effect of the "2" digit, which has "shifted" off to the right.

3. Add 9, giving 3459. This adds in the "9" digit.

Using this method results in the following calculation:

$$\begin{aligned}
3459 \ \% \ 7 &= ((2345 \times 10) - (2 \times 10000) + 9 \ \% \ 7) \ \% \ 7 \\
&= (((2345 \ \% \ 7) \times 10) - (2 \times (10000 \ \% \ 7)) + 9 \ \% \ 7) \ \% \ 7 \\
&= ((0 \times 10) - (2 \times 4) \ \% \ 7 + 9 \ \% \ 7) \ \% \ 7 \\
&= (0 - 1 + 2) \ \% \ 7 \\
&= 1
\end{aligned}$$

Similarly, once we know 3459 % 7, it is easy to use this knowledge to compute 4591 % 7, in precisely the same manner as above:

$$\begin{aligned}
4591 \ \% \ 7 &= ((3459 \times 10) - (3 \times 10000) + 1) \ \% \ 7 \\
&= (((3459 \ \% \ 7) \times 10) - (3 \times (10000 \ \% \ 7)) + 1) \ \% \ 7 \\
&= ((1 \times 10) - (3 \times 4 \ \% \ 7) + 1) \ \% \ 7 \\
&= (3 - 5 + 1) \ \% \ 7 \\
&= 6
\end{aligned}$$

8.2.3 Implementing the Rabin-Karp String Search

We will now explore how to implement this algorithm in C, for strings consisting of C characters.

A hashing function based on the function given in figure 8.1 is shown in figure 8.2. Also note that modulus m is not defined in the function; it is a parameter that can be specified at runtime.

The actual string matching algorithm is given in figure 8.3.

The first action taken is to compute b^M, which will be used later to figure out what amount to subtract from the hash value as characters are "shifted off" to the left.

To gracefully deal with the case where M is large, the divide-and-conquer algorithm for computing exponentials is used. This is the same as the algorithm you explored in the first assignment, except that the modulus is taken early and often to prevent the result from becoming too large. The code for the resulting function is shown in figure 8.4.

The next action is to laboriously hash the pattern and the substring of length M at shift zero of the text. This is the only time we'll hash a substring of the text using the hashing function; all subsequent hashes will be computed by the method described previously.

Figure 8.2: A hashing function.

```
long hash (unsigned char *str, long len, long b, long m)
{
        long i;
        long value = 0;
        long power = 1;

        for (i = len - 1; i >= 0; i--) {
                value += (power * str [i]);
                value %= m;
                power *= b;
                power %= m;
        }
        return (value);
}
```

Before we begin working our way down the string, however, we must check whether shift zero itself is a match. If it is, then we are finished. Otherwise, we loop through the rest of the string, computing each hash based on the previous (using the `nextHash` function, shown in figure 8.5).

8.2.4 Speeding Up Rabin-Karp

Avoiding the Modulo Operation

The implementation of the Rabin-Karp algorithm shown in the previous section is faithful to the original algorithm, but has a flaw that makes it somewhat suboptimal on many modern computers– it relies heavily on the modulo operation, which is usually relatively slow.[1]

However, the modulus operation can be eliminated from the algorithm by choosing a different hashing function that has the same property that it is easy to compute from the previous string, yet does not require so many modulo operations. One way to generate such a function is to keep the same basic function, but always choose a modulus that is a power of 2,

[1] Many modern compilers implement % via division on machines that do not have a `mod` instruction, and many RISC architectures do not have `mod`. Therefore, it seems that the faster your machine is, the slower % runs relative to everything else.

Figure 8.3: The Rabin-Karp String Searching Algorithm

```
long stringSearchRK (unsigned char *P, long P_len,
                unsigned char *T, long T_len,
                long b, long m)
{
        long top_one    = exp_mod (b, P_len, m);
        long P_hash     = hash (P, P_len, b, m);
        long T_hash     = hash (T, P_len, b, m);
        long i;

        if ((T_hash == P_hash) && (memcmp (T, P, P_len) == 0))
                return (0);
        for (i = 0; i <= T_len - P_len; i++) {
                T_hash = nextHash (T_hash, T[i + P_len], T[i],
                                top_one, b, m);
                if ((T_hash == P_hash) && (memcmp (T + i + 1, P, P_len) == 0))
                        return (i + 1);
        }
        return (-1);
}
```

Figure 8.4: An `exp` function that computes $x^n \% m$.

```
unsigned long exp_mod (unsigned long x, unsigned long n, unsigned long m)
{
        if (n == 0)
                return (1);
        else if (n == 1)
                return (x % m);
        else {
                unsigned long square = (x * x) % m;
                unsigned long exp = exp_mod (square, n / 2, m);

                if (n % 2 == 0)
                        return (exp % m);
                else
                        return ((exp * x) % m);
        }
}
```

Figure 8.5: The `nextHash` function.

```
/*
 * nextHash - Compute the next hash value, based on the current.  "in"
 * is the character being shifted in, "out" is the character being
 * shifted out, top_one is b^l (where l is the length of the pattern),
 * b is the base, and m the modulus.
 */
long nextHash (long curr, int in, int out, long top_one, long b, long m)
{
        curr *= b;
        curr -= ((out * top_one) % m);
        curr += in;
        curr %= m;
        if (curr < 0)
                curr += m;
        return (curr);
}
```

so that bit masks can be used instead of the more expensive % operation. Of course, using a modulus that is a power of 2 will have disastrous consequences for the hashing function if the base b used by the function is also a power of 2– which it often will be. However, we have some freedom here as well– all that we need to do is to choose a b that is relatively prime with respect to m– any number larger than the number of characters in the alphabet will do.[2] In this case, using a b of 257 is a safe bet.

A function that implements this change is shown in figure 8.6. Note that this function completely ignores its b and m parameters and instead alway uses $b = 257$ and $m = 1024$. (These parameters are retained only so this function can be used as a drop-in replacement for `stringSearchRK`.)

Avoiding False Matches Through Randomness

The Rabin-Karp algorithm, as presented here, can be very slow if the text contains a lot of "false matches": substrings that hash to the same number as the pattern and therefore cause an expensive `memcmp` to be performed. An intelligent (or lucky) adversary who knows b and m can create texts that make this algorithm perform terribly.

One solution is to choose m and b randomly at runtime, and whenever the number of false matches is high, randomly choose a new m and/or b.

[2]Using a b that is smaller than the number of characters in the alphabet is possible as well, if for some reason this is desirable, but doing so does require a bit more care.

Figure 8.6: A faster implementation of the Rabin-Karp algorithm, which avoids using the mod operation.

```
long stringSearchRKFast (unsigned char *P, long P_len,
                unsigned char *T, long T_len,
                long b, long m)
{
        long fast_b     = 257;
        long fast_m     = 1024;
        long top_one    = exp_mod (fast_b, P_len, fast_m);
        long P_hash     = hash (P, P_len, fast_b, fast_m);
        long T_hash     = hash (T, P_len, fast_b, fast_m);
        long i;

        if ((T_hash == P_hash) && (memcmp (T, P, P_len) == 0))
                return (0);
        for (i = 0; i <= T_len - P_len; i++) {
                T_hash *= fast_b;
                T_hash -= ((T[i] * top_one) & (fast_m - 1));
                T_hash += T[i + P_len];
                T_hash &= (fast_m - 1);
                if (T_hash < 0) {
                        T_hash += fast_m;
                }
                if ((T_hash == P_hash) && (memcmp (T + i + 1, P, P_len) == 0))
                        return (i + 1);
        }
        return (-1);
}
```

8.3 The Knuth-Morris-Pratt Algorithm

The Knuth-Morris-Pratt algorithm ensures that a string search will never require more than N character comparisons (once some precomputation is performed).

Imagine that we are performing a string search, and the pattern is "AAAB" and the text is "AAAXAAAAA". If we use the brute-force algorithm, then the first test will fail when the "B" in the pattern fails to match the fourth character in the text, which is an "X". At this point, the brute-force algorithm will shift the pattern by one position and start over.

However, this seems silly. In our first attempt to match by brute-force, we learned some key information about text– because we managed to successfully match three characters in the string, *we know what those three characters are and can reuse this information.* More importantly, we know implicitly what the first two characters are of the next substring that we will try– so there is no need to explicitly check them.

The Knuth-Morris-Pratt string searching algorithm uses this information to reduce the number of times it compares each character in the text to a character in the pattern. If properly implemented, the Knuth-Morris-Pratt algorithm only looks at each character in the text once.

The Knuth-Morris-Pratt algorithm, stated in terms of the brute-force algorithm, is given in algorithm 8.3.

Algorithm 8.3 (Knuth-Morris-Pratt) *Knuth-Morris-Pratt string searching*

1. Set $i = 0$, $j = 0$.
2. Match as in the brute-force algorithm, but when a mismatch is detected:
   - If $j = 0$, increment i.
   - Otherwise, let len be the length of the longest suffix of the first j characters in the pattern that appears as a prefix of the first j characters in the pattern.

     Let the *skip* be $j - len$. Increment i by the skip, and decrement j by the skip.

The basic form of the matching algorithm is very similar to the brute-force algorithm– except that instead of always incrementing `i` and setting `j` back to zero at the end of every failed substring test, we set `j` back only as far as the beginning of the possible match, and increment `i` by the corresponding amount.

The difficulty is in figuring out how far to skip when a mismatch occurs. The trick is to precompute a table of skips for each prefix ahead of time, so that in the second step of the algorithm we know immediately how much to increment i and decrement j. Note that this precomputation is based *entirely* upon P, so the cost of this precomputation is not related to the length of the text.

Before writing the code, however, we must clarify exactly what this table of skips will contain. If we have successfully matched x characters in the text against x characters in the pattern, and a mismatch is detected at the next character, how far should we skip? The goal is to skip as far as possible *without missing any potential matches*.

This turns out to be the length of the longest *prefix* of fewer than x characters in the pattern that also appears as a suffix of the first x characters in the pattern. If there is no such prefix, then we know that we can skip over x characters.

It is extremely important to note that the text itself is not considered– if we have already matched x characters in the text, then we know what that text is in terms of the pattern.

If $x = 1$, then the pattern is immaterial– the skip must always be at least one (or else we won't ever make any progress), so we abandon the matched character, skip over it, and begin again at the mismatch.

A table of the skips for several short strings is shown in figure 8.7.

8.3.1 Implementation of Knuth-Morris-Pratt

It is sometimes difficult to think concretely about this sort of problem, because humans have an enormous innate pattern-matching ability, and therefore we have the ability to use this kind of algorithm without really understanding it! Algorithms that make a great deal of intuitive sense can sometimes be difficult to implement elegantly (or even correctly). Figure 8.8 shows code that computes the skip array for a pattern.

Once the `skipArray` is computed, however, the code to actually implement the algorithm is very straightforward. A C implementation of the code to implement algorithm 8.3 is shown in figure and 8.9.

8.3.2 Analysis of Knuth-Morris-Pratt

The careful reader will note that this algorithm does not fulfill the promise that we will look at each character in the text only once. Characters in the midst of a match will only be looked at once, and characters that fail immediately to match the first character in the pattern are looked at only once, but we will look at mismatched characters twice. When a match fails somewhere in the midst of a string, the character that caused the failure is tested

Figure 8.7: The `skipArray` for several patterns. The Prefix/Suffix column shows the substring that is both a prefix and suffix of the first j characters of the pattern. A Prefix/Suffix of **nil** denotes the empty string.

Pattern	**j**	**Skip**	**Prefix/Suffix**
AABAA	2	1	A
AABAA	3	3	**nil**
AABAA	4	3	A
ABABC	2	2	**nil**
ABABC	3	2	A
ABABC	4	2	AB
AAABA	2	1	A
AAABA	3	1	A
AAABA	4	4	**nil**
ABCABC	2	2	**nil**
ABCABC	3	3	**nil**
ABCABC	4	3	A
ABCABC	5	3	AB

Figure 8.8: Code to compute the prefix lengths for the Knuth-Morris-Pratt algorithm.

```
long *skipArrayComputeKMP (unsigned char *P, long M)
{
        long *skipArray;
        long matched, len;

        if (M < 2)
                M = 2;

        skipArray = malloc (M * sizeof (long));

        /*
         * Indices 0 and 1 are unrelated to the pattern: skipArray[0]
         * should never be used, and skipArray[1] must always be 1
         * because if only one character matched before the failure,
         * the only thing we can do is skip over it-- it can't play a
         * role in a later match.
         */

        skipArray[0] = 0;
        skipArray[1] = 1;

        for (matched = 2; matched < M; matched++) {
                for (len = matched - 1; len > 0; len--) {
                        if (memcmp (P, &P[matched - len], len) == 0) {
                                break;
                        }
                }
                skipArray[matched] = matched - len;
        }
        return (skipArray);
}
```

Figure 8.9: The Knuth-Morris-Pratt algorithm, implemented in C.

```
long stringSearchKMP (unsigned char *P, long M, unsigned char *T, long N)
{
        long i, j;
        long *skipArray = skipArrayComputeKMP (P, M);
        long skip;

        j = 0;
        i = 0;
        while (i <= N - M) {
                if (j == M)
                        break;
                else if (P[j] == T[i + j])
                        j++;
                else if (j == 0)
                        i++;
                else {
                        skip = skipArray[j];
                        i += skip;
                        j -= skip;
                }
        }

        free (skipArray);

        if (i <= N - M)
                return (i);
        else
                return (-1);
}
```

again in the next attempt. Although we know what this character is and can make use of this information, the only information we are using is what it *isn't*– it isn't a match.

However, although we may have to look at some characters more than once, the algorithm looks at each character at most twice and so is $O(N)$ (see the exercises for more detail).

We can improve upon the performance of our implementation by extending our table to refine our information about the prefix lengths to include the character that caused the mismatch. This can result in a much larger table, since instead of a one-dimensional table indexed by the number of matching characters, we now need to have a two-dimensional array indexed by both the number of matching characters and the identity of the character that caused the mismatch. Although the table is often sparse, and can be represented in relatively little space, this optimization adds considerably to the memory requirements and/or complexity of the implementation. Whether the added performance is worth the extra memory and complexity is a design decision.

8.3.3 A Knuth-Morris-Pratt Example

This section illustrates the Knuth-Morris-Pratt algorithm with an example.

Let us begin with an example where the pattern is ACAC and the text is ABACABACAC. In figure 8.10, j and i are shown, along with the pattern shifted by i.

Figure 8.10: An illustration of the Knuth-Morris-Pratt algorithm.

```
Text    = ABACABACAC
Pattern = ACAC

          ABACABACAC            i = 0
          ACAC                  j = 0
          ^                     match
          ABACABACAC            i = 0
          ACAC                  j = 1
           ^                    mismatch! skip 1
          ABACABACAC            i = 1
           ACAC                 j = 0
           ^                    mismatch! skip 1
          ABACABACAC            i = 2
            ACAC                j = 0
            ^                   match
          ABACABACAC            i = 2
            ACAC                j = 1
             ^                  match
          ABACABACAC            i = 2
            ACAC                j = 2
              ^                 match
          ABACABACAC            i = 2
            ACAC                j = 3
               ^                mismatch! skip 2
          ABACABACAC            i = 4
              ACAC              j = 0
              ^                 match
          ABACABACAC            i = 4
              ACAC              j = 1
               ^                mismatch! skip 1
          ABACABACAC            i = 5
               ACAC             j = 0
               ^                mismatch! skip 1
          ABACABACAC            i = 6
                ACAC            j = 0
                ^               match
  (Final four matches omitted)  MATCH at i = 6
```

8.4 The Boyer-Moore Algorithm

Once again, imagine that we are performing a string search, and the pattern is "AAAB" and the text is "AAAXAAAB".

If we skip ahead and look at the fourth character of the text, we can immediately note that this character appears nowhere in the pattern. Therefore, if there is a match anywhere in this text, *it cannot possibly overlap with the fourth character in the text.* Since the pattern is of length 4, that means that the smallest possible shift that needs to be considered is 4. Therefore, just by checking a single character, we have completely eliminated the need to look at the first three characters in the text!

This kind of skip is the goal of the Boyer-Moore algorithm– unlike the Knuth-Morris-Pratt algorithm, which strives to look at each character in the text only once, the Boyer-Moore algorithm strives to completely ignore as many characters in the text as possible. The algorithm itself is given in algorithm 8.4.

Once again, the algorithm is very simple, but the computation of the skips is not. As we did in the Knuth-Morris-Pratt algorithm, we will precompute an array of skips, but unlike the Knuth-Morris-Pratt algorithm, this array will be two-dimensional– it will take into account both the number of characters successfully matched (j) and the identity of the character in the text (c) that caused the mismatch.

It is much easier to understand the computation of the skips from an intuitive perspective and then see how this algorithm obeys our intuition than it is to prove that this algorithm is correct and complete, so that is how I will proceed.

In the calculation of the skips, there are two basic cases that we need to consider in algorithm 8.4:

1. For $d \leq j$, then the new shift must overlap all of the pattern that we have already successfully matched. Therefore, it must match all of the pattern we have already matched (because otherwise, a mismatch is inevitable). In addition, the character that caused the mismatch in the current shift must match its counterpart in the pattern in the new shift (because, once again, otherwise we know that a mismatch is inevitable if this character does not match the new shift of the pattern).

2. If $d > j$, then we are shifting "past" the part of the pattern that we have already matched– we are shifting far enough so that the new shift moves past some or all of the characters already matched.

   We have determined that the mismatched character cannot appear anywhere among the characters we have already matched, because there is no shift that contains it and

Algorithm 8.4 (Boyer-Moore) *Boyer-Moore string searching.*

Let P be the pattern and M its length. Let T be the pattern and N its length.
Let i be the shift, which is initially zero. Execute the following loop, until a match has been found or the end of the text is reached (when $i = N - M$). (If the end of the text is reached, return -1 to indicate failure.)

1. Attempt to match the pattern to the text at shift i, starting from the end of the pattern and moving towards the front. If the pattern completely matches the text at this shift, return i. Otherwise, let j represent the index in the pattern where the mismatch occured. (The number of successfully matched characters is therefore $M - j$.)

2. Advance i by d positions. The skip d is chosen to be the smallest d that satisfies the following conditions:

   Let **start** $= \mathbf{max}(j + 1, d)$, and let **len** $= M -$ **start**.

   - $d \leq j$ and the substring of length **len** starting at offset **start** $- d$, in P matches the substring starting at offset **start**, and $P[j - d] == c$.
   - $d > j$ and the substring of length d starting at offset 0 in P matches the substring of length **len** that starts at offset **start** in P.

would be consistent with the other characters we have already matched– if there was, we would have already found it when we were investigating the first case. Therefore, we must skip past the characters we have already matched in the text, looking for a way to match them up with the pattern.

The only question is how far to skip– and this is determined, in a similar manner as the Knuth-Morris-Pratt algorithm, by finding the length of the longest suffix of the matched characters that matches a prefix of the pattern.

Note that once again, we always choose the smallest skip that has not already been proven to lead to a mismatch. This prevents the mishap of skipping past a match.

8.4.1 Implementation of Boyer-Moore

The code that implements the heart of the Boyer-Moore algorithm is shown in figure 8.11, and the code that implements the computation of the skip array is shown in figure 8.12.

As in the implementation of the Knuth-Morris-Pratt algorithm, no special effort is taken to make sure that this implementation is efficient– in particular, the algorithm used to compute the skip array is quite inefficient. It is possible to compute the skip array in time $O(M)$ (rather than $O(M^3)$ as this code does), but the algorithm for doing so is quite complex.[3] This means that for long patterns or short texts, this implementation is quite a bit slower than any of the other algorithms shown in this chapter. However, once the text length grows large, the true speed of this algorithm becomes apparent– for texts of more than a few hundred kilobytes, this algorithm is roughly twice as fast as any of the others.[4] Clearly, when the text is long, and speed is important, an optimized Boyer-Moore function will give superior performance.

8.4.2 A Boyer-Moore Example

This section illustrates the Boyer-Moore algorithm with an example.

Let us begin with an example where the pattern is ACAC and the text is ABACABACAC. In figure 8.13, j and i are shown, along with the pattern shifted by i:

[3] It would make a **wonderful** (and ambitious) final project for someone to nail down the $O(M)$ algorithm for computing the skip array.

[4] Using randomly generated text and pattern strings.

Figure 8.11: Code that implements the Boyer-Moore algorithm.

```
long stringSearchBM (unsigned char *P, long M, unsigned char *T, long N)
{
	long i, j;

	skipArrayComputeBM (P, M);

	i = 0;
	while (i <= N - M) {
		j = M - 1;
		while ((j >= 0) && (P[j] == T[i + j]))
			j--;
		if (j < 0)
			break;
		i += skipArray[j][T[i + j]];
	}
	if (i <= N - M)
		return (i);
	else
		return (-1);
}
```

Figure 8.12: Code to compute the skipArray for the Boyer-Moore algorithm.

```
void skipArrayComputeBM (unsigned char *P, long M)
{
        long c, j;

        for (j = 0; j < M; j++)
                for (c = 0; c < ALPHA_SIZE; c++)
                        skipArray [j][c] = prefixComputeBM (P, M, j, c);
        return ;
}

long prefixComputeBM (unsigned char *P, long M, long j, long c)
{
        long d, start, len;

        for (d = 1; d < M; d++) {
                if (d <= j) {
                        start = j + 1;
                        len = M - start;

                        if ((memcmp (&P[start - d], &P[start], len) == 0) &&
                                        (P[j - d] == c)) {
                                return (d);
                        }
                }
                else {
                        len = M - d;
                        if (memcmp (P, &P[d], len) == 0)
                                return (d);
                }
        }
        return (d);
}
```

Figure 8.13: An illustration of the Boyer-Moore algorithm.

```
Text    = ABACABACAC
Pattern = ACAC

          ABACABACAC            i = 0
          ACAC                  j = 3
             ^                  match

          ABACABACAC            i = 0
          ACAC                  j = 2
            ^                   match

          ABACABACAC            i = 0
          ACAC                  j = 1
           ^                    mismatch! skip 2

          ABACABACAC            i = 2
            ACAC                j = 3
               ^                mismatch! skip 4

          ABACABACAC            i = 6
                ACAC            j = 3
                   ^            match

          ABACABACAC            i = 6
                ACAC            j = 2
                  ^             match

          ABACABACAC            i = 6
                ACAC            j = 1
                 ^              match

          ABACABACAC            i = 6
                ACAC            j = 0
                ^               match
                                MATCH at i = 6
```

8.5 Exercises

1. The Rabin-Karp algorithm can be used very effectively for 2-dimensional pattern matching– finding whether a "pattern" array of characters appears as a subarray of a "text" array of characters.

   Show how the algorithm could be changed for the two-dimensional case, and generalize your algorithm to work with higher numbers of dimensions.

2. Find at least one other hashing function that can be used with the Rabin-Karp string matching algorithm. Try to find one that is efficient to compute.

3. Modify the `stringSearchRK` function so that it randomly selects a new prime m whenever the number of mismatches exceeds $4/m$ of the total number of characters processed at any point in the text. You may assume the existence of an array of suitable primes.

4. What is the maximum number of times that any character in the text will be looked at by the `stringSearchKMP` function? Is it possible that adjacent characters in the text will be considered this many times?

5. The way that the `skipArray` is computed for the Knuth-Morris-Pratt string searching algorithm is quite inefficient. This means that searching for relatively long patterns in relatively short texts is inefficient with this function, because the time necessary to compute the `skipArray` is comparable to the time necessary to perform the search itself!

   Show how the computation of the `skipArray` can be significantly improved. (It can be done in time $O(M)$.) This is not an easy problem.

Appendix A

Sparse Arrays Revisited

As mentioned at the end of chapter 3, there are a number of ways to represent sparse arrays. Which representation is best depends largely on how the arrays will be used.

Now that we have had a chance to see several other data structures, it makes sense to revisit the topic of sparse arrays and see how these new data structures can be used in this context.

A.1 Hash Tables

A hash table is a natural way to represent an array. (In fact, several interpreted languages implement all of their arrays as hash tables.)

The indices of the element being accessed are used as the argument to the hashing function. Once the hash bucket where the element resides has been determined, the bucket is searched for the element with the correct coordinates. If the element is missing, then it is assumed to have the default value.

The benefit of this approach is that if the hash function is effective (i.e. each hash bucket doesn't have too many elements in it, so that the search is fast), then accessing an element in the array is effectively O(1).

However, achieving this level of performance is not simple.

First, the hashing function must be chosen so that is relatively unlikely that a large number of the array elements will hash to the same bucket– for sparse matrices, this is not usually a problem (since there are relatively few interesting elements to begin with), but as the array becomes less sparse, the size of the hash table must grow. Eventually, it may become large enough so that there is no savings realized by using the hash table. In addition, if the pattern of the non-zero elements in the sparse array triggers degenerate behavior in the hash function, the hash table may become quite slow.

Second, the hashing function must be quick to compute. Even though a hash table can (at least in theory) implement array element access operations in $O(1)$ time, the constant overhead is much higher than that of ordinary array element access.

One large problem with the hash table representation is the lack of information about neighboring elements that can be easily derived from an element. Using the linked list method, it is easy to progress from non-zero element to non-zero element– just a matter of traversing along the linked list. In a hash table, the closest thing would be traversing through a bucket– but the elements in a particular bucket will typically represent elements scattered throughout the array, and there is generally no useful meaning to the order of the elements in the bucket.

A.2 Binary Search Trees

Instead of using linked lists to represent the rows and/or columns of sparse arrays, we can use binary search trees. If the trees are kept balanced, then this means that array element access can be performed in $O(\log x)$ time (where x is the number of non-zero elements in the given row or column). With some care, even a completely full n-by-n array can have its elements accessed in $O(\log n)$ time. While this is quite a bit more than the $O(1)$ required by the ordinary representation, it is also much less than the $O(n)$ required in the worst case by the linked-list approach.

Implementing binary trees instead of linked lists adds a degree of difficulty to the representation– keeping trees balanced is challenging, and traversing the tree inorder (to achieve the same effect as traversing the linked list) is somewhat more so. However, these problems have all been solved before, so they should present no overwhelming obstacles.

A.3 Exercises

1. Implement a library of functions to implement sparse arrays as hash tables. Make sure that you address the issue of finding a hash function that works well for arrays of different sizes.

2. Devise an efficient algorithm that finds the intersection of two binary search trees (i.e. the list of keys that appear in both trees) and prints it in order. The algorithm must visit each node in both trees *at most* once.

3. Using the algorithm from the previous exercise, implement the fast matrix multiplication algorithm for sparse arrays described at the end of chapter 3, modified to work with sparse arrays represented by binary trees.

4. Devise an efficient algorithm that finds the union of two binary search trees (i.e. the list of keys that appear in either trees) and prints it in order.

Appendix B

Pointers and Malloc

For a more complete treatment of this material, refer to your favorite C reference.

B.1 Memory and Pointers

To be blunt, most of this section should be a review. If much of the material in this section is completely new to you, then there is a good chance that you are not prepared to take S-Q. The majority of the things we'll do in this course require extensive use of pointers, and this material must be understood in order for you to complete the assignments.

B.1.1 Memory and Addresses

C treats memory as a large array of bytes. Every piece of memory that a program uses actually resides somewhere in this array.

The *address* of a location of memory is the index of its beginning in this "memory array". It is often useful to refer to a variable or other object by this location, instead of by its name. In fact, this is the only possible way to deal with dynamically allocated objects (discussed in section B.3.1), which do not have names in the usual sense.

A *pointer* is a C term for an address; pointers and addresses are synonymous, even though one or the other is usually used in particular idioms. (This may seem confusing at first, but after a while you won't even hear the difference.)

B.1.2 C Syntax for Pointers

C has two operators for working with pointers.

& *Take the address of* or *get a pointer to* some location in memory.

***** *Deference an address* or *follow the pointer*. Fetch the value stored at the given address.

These are both *unary* operators (i.e. they have one operand), and they come before the expression that they operate upon.

In C, pointers have specific types. The type of a pointer is that of a pointer to the type of thing that it can point to. This type is specified by following the type pointed to with an asterix (*).

```
int i;          /* i is an ordinary int.       */
int *ip;        /* ip is of type "int *".  It  */
                /* is a pointer to an int.     */
int **ipp;      /* ipp is of type "int **"; it */
                /* is a pointer to a pointer   */
                /* to an int.                  */
```

Be careful, however, because the asterix in one of these types is associated with the variable, not the type. For example:

```
int i;          /* i is an ordinary int.       */
int *ip1, ip2;  /* ip1 is of type "int *", but */
                /* ip2 is an ordinary int.     */
```

For this reason, most C programmers place the asterix in a pointer type at the start of the variable name, rather than the end of the type, in order to avoid confusion. Instead of writing "`int* ip`", which makes it look like the `int` and the asterix are somehow bound together, they would write "`int *ip`".

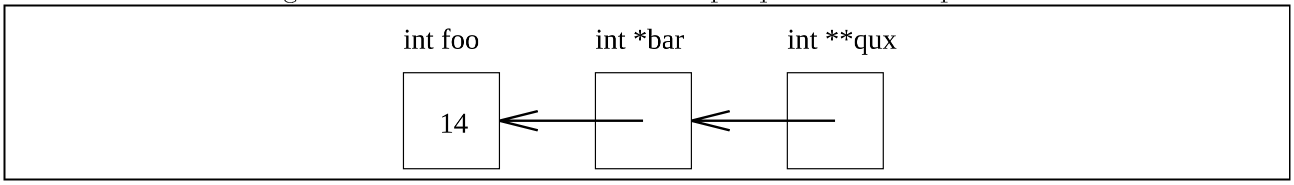
The exception to the strict typing of pointers is the *void pointer* type, which can point to something of any type.

Pointer casts can be used to override these types (in a manner similar to the way that type casts can be used to convert integers into doubles and vice versa). Pointer casts are generally something to avoid, except in the case of a void pointer. A `void` pointer must be cast (or assigned to a variable of a specific variable type) before it can be dereferenced.

Figure B.1: A simple pointer example.

```
int foo;          /* foo is an ordinary int. */
int *bar;         /* bar is a pointer to an int. */
int **qux;        /* qux is a pointer to a pointer to an int */

foo = 14;         /* foo gets value 14. */
bar = &foo;       /* bar points to foo. */
qux = &bar;       /* qux points to bar. */
```

Figure B.2: The result of the simple pointer example.



As an example of the use of pointers, consider the code shown in figure B.1. After this code is executed, the result is illustrated by figure B.2.

With these pointers set up, there are now several ways that we can access the value stored in the variable `foo`. We can use `foo` itself, as always; the fact that other variables point to `foo` does not limit the way that we use `foo` in any manner.

```
foo = 12;        /* assigns the value 12 to foo. */

printf ("%d\n", foo); /* prints the value of foo*/
```

We can also access the value stored in `foo` via the pointer from `bar` to `foo`:

```
*bar = 12;       /* assigns the value 12 to      */
                 /* whatever bar points to.      */
printf ("%d\n", *bar); /* prints the value of   */
                 /* whatever bar points to.      */
```

Since `qux` points to `bar`, and `bar` points to `foo`, we also can deref `qux` twice in order to access `foo`:

```
    **qux = 12;     /* assigns the value 12 to      */
                    /* whatever the pointer qux     */
                    /* points to points to.         */
    printf ("%d\n", **qux); /* prints the value of  */
                    /* whatever the pointer qux     */
                    /* points to points to.         */
```

B.1.3 Call by Reference versus Call by Value

In general, C functions use *call by value* for passing their arguments: the *value* of expressions used as arguments to functions are passed to the functions, and not the expressions themselves.

For example:

```
void foo (int y)
{
    y = 5;
    printf ("y inside foo = %d\n", y);
    return ;
}

int main ()
{
    int x = 10;

    printf ("x inside main = %d\n", x);
    foo (x);
    printf ("x inside main = %d\n", x);
    return ;
}
```

In this example, when `main` calls `foo`, the value of `x`, which is 10, is passed to `foo`, and winds up in `foo`'s own copy of `y`. If `foo` changes the value of its `y`, the value of `x` inside `main` is unchanged.

When this program is run, it will produce the output:

```
x inside main = 10
y inside foo = 5
x inside main = 10
```

However, pointers allow you to sidestep call by value. Instead of passing a copy of the value to the function, you can pass a reference (or pointer) to the variable to the function. The function can then dereference the pointer, and change the variable as it pleases. This way of passing parameters is called call by reference. For example:

```
void foo (int *y)
{
    *y = 5;
    printf ("*y inside foo = %d\n", *y);
    return ;
}

int main ()
{
    int x = 10;

    printf ("x inside main = %d\n", x);
    foo (&x);
    printf ("x inside main = %d\n", x);
    return ;
}
```

This program will produce the following output:

```
x inside main = 10
*y inside foo = 5
x inside main = 5
```

B.2 Arrays and Pointers in C

In C, there are strong relationships between pointers and arrays. For any one-dimensional array `A` and integer `x`, the following relationships are always true:

1. `&A[0] == A`

   The address of the first element of the array `A` is value of `A` itself. In other words, `A` is a pointer, it points to the first element in the array.

2. `&A[x] == A + x`

   The address of the x'th element of `A` is the value of `A + x`.

   This is one of the great truths (and a defining characteristic) of the C programming language. The first relationship is a special case of this more general relationship.

   This equality defines part of the nature of *pointer arithmetic*, which defines how the addition operator works when it is computing the sum of a pointer and an integer. Instead of doing ordinary arithmetic (treating `A` as though it was an ordinary integer and adding `x` to it), C takes into consideration the size (in bytes) of the type pointed to by `A`. The actual arithmetic performed (behind your back by the compiler) is to compute `A + (x * sizeof(t)`, where `t` is the type pointed to by `A`.

   For example, if `A` points to address 1000, and is a pointer to an `int`, then `A + 1` evaluates to a pointer to address 1004 on the course machines (because on the course machines, an `int` consists of 4 bytes). Similarly, if `B` points to address 2000, and is a pointer to a `char`, then `B + 3` evaluates to 2003 (because on the course machines, a `char` consists of a single byte).

   (Note that you can also subtract an integer from a pointer, which allows you to index backwards off the front of an array. For example, `A - 1` gives a pointer to the "-1'th" element of `A`. There are precious few situations where it makes any sense to do this, but it can be useful.)

3. `A[x] == *(A + x)`

   This is basically the same as the previous relationship, but adds the reassuring fact that this relationship still holds if we dereference both sides of the equality.[1]

4. `(&A[x] - &A[y]) == (x - y)`

   This relationship defines subtraction of pointers. The subtraction of two pointers to type t is the number of elements of type t that would fit between them.

   For example, if `A` is an array of `int`s, starting at address 5000, and `x` is 10 and `y` is 4, then then (again on the course machines, where the size of an `int` is 4) `&A[x] == 5040` and `&A[y] == 5016`. The difference between 5040 and 5016 is 24. Dividing 24 by the size of an `int` gives 6, which is indeed `x - y`.

[1] It would be strange if this was not the case, but language designers have been known to do strange things.

> Note that if the two pointers being subtracted are not of the same type, the result of the subtraction is not defined. Most C compilers will not permit you to do this, but by abusing pointer casts, you can get most compilers to let you– but when you do this, the bugs that may result are your own fault.

These relationships allow us to treat arrays like pointers, and treat pointers like arrays. Both of these treatments can be very useful, particularly when dealing with dynamically allocated memory, as we will see in the next section.

B.3 Dynamic Memory Allocation

In many cases, it is impossible to know (or impractical to determine) when a C program is written and compiled, how much memory will be used by the program when it executes.

One approach to this problem is to have the programmer figure out, when he or she is writing the program, how much space it will require (or impose arbitrary limits on how much space the program can use), and write their code accordingly. This approach is insufficient for many common kinds of programs, however. For example, imagine how wasteful it would be if your word processor were written in such a way that it always grabbed enough memory so that it could edit an encyclopedia, even if all you ever edited with it was your grocery list.

Dynamic memory allocation is a way to defer the decision of how much memory is necessary until the program is actually running, get more if you run out, or give back memory that your program no longer needs.

B.3.1 Malloc, Sizeof and Free

Malloc

The `malloc` (*m*emory *alloc*ation) function is the most frequently used function for dynamically allocating memory. `malloc` has the following prototype:

```
void *malloc(unsigned int size);
```

The `size` parameter indicates the number of bytes of memory that you wish to allocate. If there is sufficient memory available, then `malloc` returns a pointer to the start of a span of memory large enough to hold `size` bytes. If sufficient memory is not available, the `malloc` returns a `NULL` pointer. (`NULL` is a constant defined as part of ANSI C, and its actual definition resides in `stdio.h`. If you try to compile a file and you get error messages about `NULL` not being defined, check to make sure that you are `#include`'ing `stdio.h`.)

Because `malloc` can return `NULL` instead of a usable pointer, your code should *always* check the return value of `malloc` in order to see whether it was successful– because if it wasn't, and your program dereferences the resulting `NULL` pointer, your program will crash. In the example code given here, I'll usually omit this check in order to keep the code short and uncluttered– but in your own code, don't neglect this check.

Also note that memory allocated by `malloc` has undefined values– there's no telling what values reside in those bytes. It is a mistake to assume that the memory is filled with zeros or some other value. Most of the time when you use `malloc`, the first thing that you will do is initialize all of the memory you allocated to some known values. (Because zero is the most common initial value, another function named `calloc` also exists in the standard library. `calloc` is similar to `malloc`, but takes different arguments; see its manual page for more info.)

Sizeof

Calculating how many bytes are necessary to store items of different types is made simpler by the `sizeof` operator. The `sizeof` operator evaluates to the number of bytes necessary to represent a specific C type or variable. For example, on the HP machines:

```
sizeof (char)   == 1
sizeof (int)    == 4
sizeof (double) == 8
```

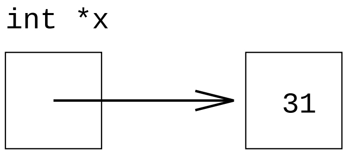
Even if you know what the size of something is, it is considered very poor style to use that number instead of relying on `sizeof`, because this makes porting your code from one platform to another difficult– and one tiny error can have strange and hard-to-diagnose results.

The return type of `malloc` is a `void` pointer. A `void` pointer may point to an object of any type, and therefore C allows a `void` pointer to be assigned to a pointer variable of any type.

For example, imagine that you wanted to allocate an `int`, and then assign 31 to this new integer:

```
int *x = malloc (sizeof (int));
*x = 31;
```

This results in the following arrangement:

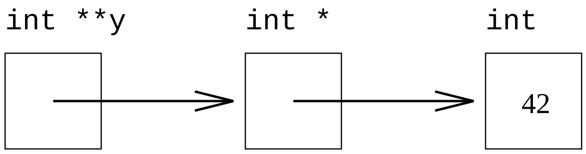


Note that this new `int` does not have a name; the only way we can refer to it is as the `int` that `x` points to. (Of course, we could create other pointers that pointed to the same place that `x` does, but in this code fragment `x` is the only pointer we have).

As a further (and more farfetched) example, consider the following:

```
int **y = malloc (sizeof (int *));
*y = malloc (sizeof (int));
**y = 42;
```

This results in the following arrangement:



Note again that the `int` pointer pointed to by `y` has no name, nor does the `int` it points to. The only way to reference this new `int` at this point is through two levels of indirection.

Free

When your program is finished using whatever memory it dynamically allocates, it can use the `free` function to tell the system that the memory is available again.

Take care to only `free` pointers that were returned from `malloc` (or one of its relatives); otherwise, you might free some memory that you didn't allocate, deallocating memory that might still be in use by some other part of your program. Many modern versions of `free` are intelligent enough to recognize when the program tries to free a bogus pointer, but not all of them do.

B.4 Dynamically Allocating Arrays

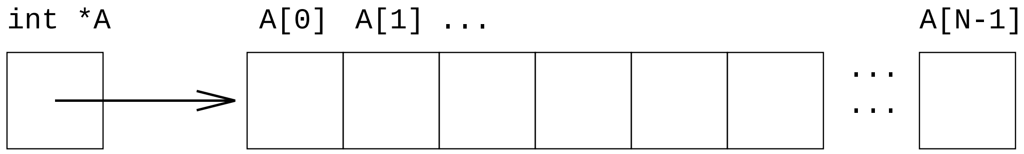
B.4.1 One-Dimensional Arrays

In C, arrays must have their extents defined at compile-time. There's no way to postpone the definition of the size of an array until runtime. Luckily, with pointers and `malloc`, we can work around this limitation.

To allocate a one-dimensional array of length N of some particular type, simply use `malloc` to allocate enough memory to hold N elements of the particular type, and then use the resulting pointer as if it were an array. For example, the following code snippet allocates a block of N `int`s, and then, using array notation, fills it with the values 0 through $N-1$:

```
int *A = malloc (sizeof (int) * N);
int i;

for (i = 0; i < N; i++)
    A[i] = i;
```

Pictorially, the "array" `A` might be drawn as:



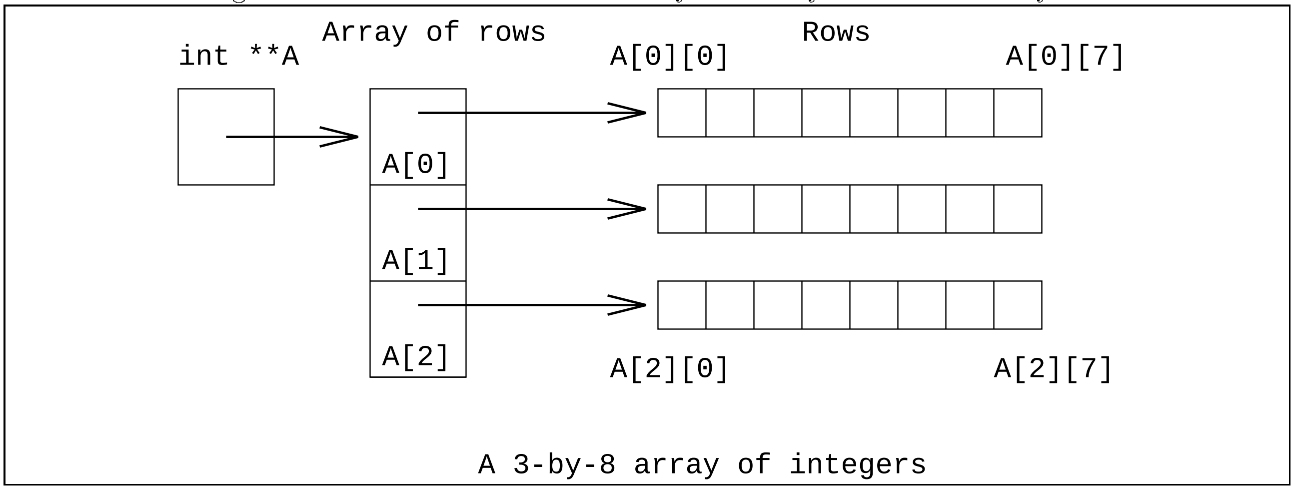
This idea is very useful for dealing with strings, which in C are represented by arrays of `char`s, terminated with a `'\0'` character. These arrays are nearly always expressed as pointers in the declaration of functions, but accessed via C's array notation. For example, here is a function that implements `strlen`:

```
int strlen (char *s)
{
    int i;

    for (i = 0; s[i] != '\0'; i++)
        ;

    return (i)
}
```

B.4.2 Two-Dimensional Arrays

Unfortunately, the convenient relationship between pointers and arrays in C breaks down for arrays of higher dimension. Luckily, we can treat two-dimensional arrays as arrays of one-dimensional arrays. An illustration of this is shown in figure B.3, and code to allocate such an array is shown in figure B.4.

Figure B.3: A Two-Dimensional Dynamically Allocated Array



A Digression Into Engineering

From the perspective of software engineering, the code in figure B.4 has serious flaws. First, it doesn't do *any* error checking– it doesn't check to make sure that the number of rows and columns requested are sensible, and it doesn't check any of the values returned from `malloc` to make sure that they are not `NULL`. Next, the array returned is filled with undefined values– whatever values happened to be in the memory that `malloc` allocated for you are now the contents of this array. Finally, the array itself carries around no information about what its extents are (how many rows and columns it has). This information may be represented elsewhere in the program, but general software engineering guidelines recommend that this information be kept with the array.

Figure B.5 shows a more realistic approach, which defines a new type named `array2d_t` to represent a dynamically allocated 2-d array, and a function to create arrays and fill them with a specific initial value.

Figure B.4: Code to create an array.

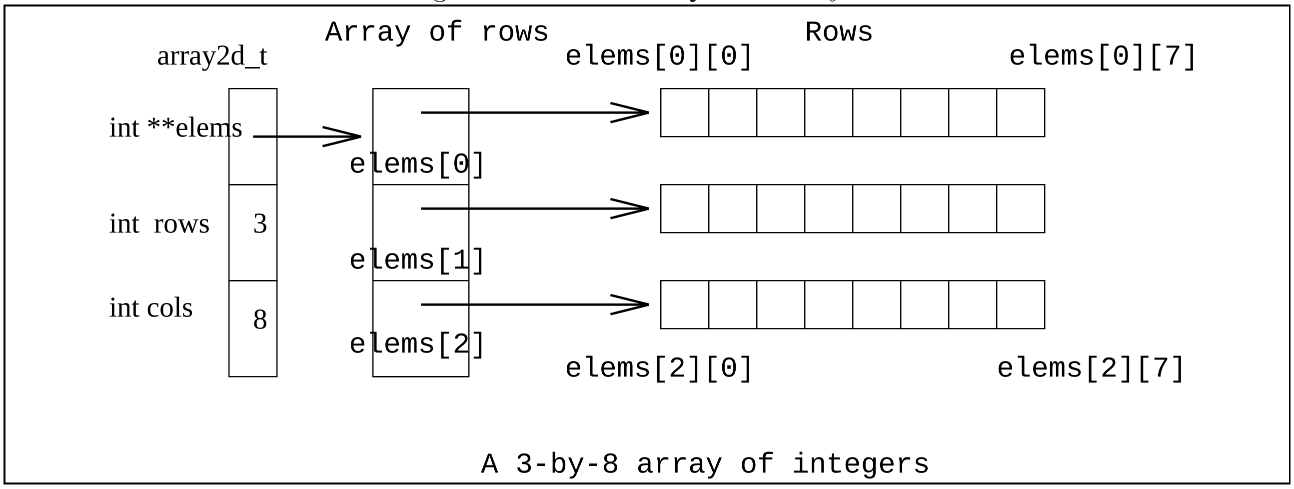
```
int **arrayCreate (int rows, int cols)
{
        int i;
        int **A;

                /* Allocate the array of pointers to rows. */
        A = malloc (rows * sizeof (int *));

                /* Allocate each row. */
        for (i = 0; i < rows; i++) {
                A [i] = malloc (cols * sizeof (int));
        }

        return (A);
}
```

An illustration of the result of this code, again for allocating a 3-by-8 array, is shown in figure B.6.

Figure B.5: Better code to allocate a 2-d array.

```
typedef struct  {
        int     **elems;
        int     rows, cols;
} array2d_t;

array2d_t *array2dCreate (int rows, int cols, int initial_value)
{
        array2d_t *array;
        int i, j;

        if ((rows <= 0) || (cols <= 0))
                return (NULL);
        array = malloc (sizeof (array2d_t));
        if (array == NULL)
                return (NULL);
        array->rows = rows;
        array->cols = cols;
        array->elems = malloc (rows * sizeof (int *));
        if (array->elems == NULL) {
                free (array);
                return (NULL);
        }
        for (i = 0; i < rows; i++) {
                array->elems [i] = malloc (cols * sizeof (int));
                if (array->elems [i] == NULL) {
                        for (j = 0; j < i; j++)
                                free (array->elems [j]);
                        free (array->elems);
                        free (array);
                        return (NULL);
                }
                for (j = 0; j < cols; j++)
                        array->elems [i][j] = initial_value;
        }
        return (array);
}
```

Figure B.6: An `array2d_t` array.



Appendix C

Data Representation

In order to understand how a computer is able to manipulate data and perform computations, you must first understand how data is represented by a computer.

At the lowest level, the indivisible unit of data in a computer is a *bit*. A bit represents a single binary value, which may be either 1 or 0. In different contexts, a bit value of 1 and 0 may also be referred to as "true" and "false", "yes" and "no", "high" and "low", "set" and "not set", or "on" and "off".

The decision to use binary values, rather than something larger (such as decimal values) was not purely arbitrary– it is due in a large part to the relative simplicity of building electronic devices that can manipulate binary values.

C.1 Representing Integers

C.1.1 Unsigned Binary Numbers

While the idea of a number system with only two values may seem odd, it is actually very similar to the decimal system we are all familiar with, except that each digit is a bit containing a 0 or 1 rather than a number from 0 to 9. (The word "bit" itself is a contraction of the words "binary digit") For example, figure C.1 shows several binary numbers, and the equivalent decimal numbers.

In general, the binary representation of 2^k has a 1 in binary digit k (counting from the *right*, starting at 0) and a 0 in every other digit. (For notational convenience, the ith bit of a binary number A will be denoted as A_i.)

The binary representation of a number that is not a power of 2 has the bits set corre-

Figure C.1: Binary and Decimal Numbers

Binary		**Decimal**
0	=	0
1	=	1
10	=	2
11	=	3
100	=	4
101	=	5
110	=	6
⋮	⋮	⋮
11111111	=	255

sponding to the powers of two that sum to the number: for example, the decimal number 6 can be expressed in terms of powers of 2 as $1 \times 2^2 \ + \ 1 \times 2^1 \ + \ 0 \times 2^0$, so it is written in binary as 110.

An eight-digit binary number is commonly called a *byte*. In this text, binary numbers will usually be written as bytes (i.e. as strings of eight binary digits). For example, the binary number 101 would usually be written as 00000101– a 101 padded on the left with five zeros, for a total of eight digits.

Whenever there is any possibility of ambiguity between decimal and binary notation, the *base* of the number system (which is 2 for binary, and 10 for decimal) is appended to the number as a subscript. Therefore, 101_2 will always be interpreted as the binary representation for five, and never the decimal representation of one hundred and one (which would be written as 101_{10}).

Conversion of Binary to Decimal

To convert an unsigned binary number to a decimal number, add up the decimal values of the powers of 2 corresponding to bits which are set to 1 in the binary number. Algorithm C.1 shows a method to do this. Some examples of conversions from binary to decimal are given in figure C.2.

Since there are 2^n unique sequences of n bits, if all the possible bit sequences of length n are used, starting from zero, the largest number will be $2^n - 1$.

Algorithm C.1 (Binary to Decimal) *To convert a binary number to decimal.*

- Let X be a binary number, n digits in length, composed of bits $X_{n-1} \cdots X_0$.
- Let D be a decimal number.
- Let i be a counter.

1. Let $D = 0$.
2. Let $i = 0$.
3. While $i < n$ do:
    - If $X_i == 1$ (i.e. if bit i in X is 1), then set $D = (D + 2^i)$.
    - Set $i = (i + 1)$.

Figure C.2: Examples of Conversion from Binary to Decimal

Binary						**Decimal**
00000000	=	0	=	0	=	0
00000101	=	$2^2 + 2^0$	=	$4 + 1$	=	5
00000110	=	$2^2 + 2^1$	=	$4 + 2$	=	6
00101101	=	$2^5 + 2^3 + 2^2 + 2^0$	=	$32 + 8 + 4 + 1$	=	45
10110000	=	$2^7 + 2^5 + 2^4$	=	$128 + 32 + 16$	=	176

Conversion of Decimal to Binary

An algorithm for converting a decimal number to binary notation is given in algorithm C.2.

Algorithm C.2 (Decimal to Binary) *To convert a positive decimal number to binary.*

- Let X be an unsigned binary number, n digits in length.
- Let D be a positive decimal number, no larger than $2^n - 1$.
- Let i be a counter.

1. Let $X = 0$ (set all bits in X to 0).
2. Let $i = (n - 1)$.
3. While $i \geq 0$ do:
   (a) If $D \geq 2^i$, then
      - Set $X_i = 1$ (i.e. set bit i of X to 1).
      - Set $D = (D - 2^i)$.

   (b) Set $i = (i - 1)$.

Addition of Unsigned Binary Numbers

Addition of binary numbers can be done in exactly the same way as addition of decimal numbers, except that all of the operations are done in binary (base 2) rather than decimal (base 10). Algorithm C.3 gives a method which can be used to perform binary addition.

When algorithm C.3 terminates, if c is not 0, then an *overflow* has occurred– the resulting number is simply too large to be represented by an n-bit unsigned binary number.

C.1.2 Signed Binary Numbers

The major flaw with the representation that we've used for unsigned binary numbers is that it doesn't include a way to represent negative numbers.

Algorithm C.3 (Unsigned Binary Addition) *Addition of unsigned binary numbers.*

- Let A and B be a pair of n-bit binary numbers.
- Let X be a binary number which will hold the sum of A and B.
- Let c and $\hat{c}$ be carry bits.
- Let i be a counter.
- Let s be an integer.

1. Let $c = 0$.
2. Let $i = 0$.
3. While $i < n$ do:
   (a) Set $s = A_i + B_i + c$.
   (b) Set X_i and $\hat{c}$ according to the following rules:
      - If $s == 0$, then $X_i = 0$ and $\hat{c} = 0$.
      - If $s == 1$, then $X_i = 1$ and $\hat{c} = 0$.
      - If $s == 2$, then $X_i = 0$ and $\hat{c} = 1$.
      - If $s == 3$, then $X_i = 1$ and $\hat{c} = 1$.

   (c) Set $c = \hat{c}$.
   (d) Set $i = (i + 1)$.

There are a number of ways to extend the unsigned representation to include negative numbers. One of the easiest is to add an additional bit to each number that is used to represent the *sign* of the number– if this bit is 1, then the number is negative; otherwise the number is positive (or vice versa). This is analogous to the way that we write negative numbers in decimal– if the first symbol of the number is a negative sign, then the number is negative, otherwise the number is positive.

Unfortunately, when we try to adapt the algorithm for addition to work properly with this representation, this apparently simple method turns out to cause some trouble. Instead of simply adding the numbers together as we do with unsigned numbers, we now need to consider whether the numbers being added are positive or negative. If one number is positive and the other negative, then we actually need to do subtraction instead of addition, so we'll need to find an algorithm for subtraction. Furthermore, once we've done the subtraction, we need to compare the the unsigned magnitudes of the numbers to determine whether the result is positive or negative!

Luckily, there is a representation that allows us to represent negative numbers in such a way that addition (or subtraction) can be done easily, using algorithms very similar to the ones that we already have. The representation that we will use is called *two's complement* notation.

To introduce two's complement, we'll start by defining, in algorithm C.4, the algorithm that is used to compute the negation of a two's complement number.

Algorithm C.4 (Two's Complement Negation) *Negation of a two's complement number.*

1. Let $\bar{x}$ = the *logical complement* of x.

   The logical complement (also called the *one's complement*) is formed by flipping all the bits in the number, changing all of the 1 bits to 0, and vice versa.

2. Let $X = \bar{x} + 1$.

   If this addition *overflows*, then the overflow bit is discarded.

By the definition of two's complement, $X \equiv -x$.

Figure C.3 shows the process of negating several numbers. Note that the negation of zero is zero.

This representation has several important properties:

Figure C.3: Examples of Negation Using Two's Complement

	00000110	=	6
1's complement	11111001		
Add 1	11111010	=	-6
	11111010	=	-6
1's complement	00000101		
Add 1	00000110	=	6
	00000000	=	0
1's complement	11111111		
Add 1	00000000	=	0

- The leftmost (most significant) bit also serves as a sign bit; if 1, then the number is negative, if 0, then the number is positive or zero.
- The rightmost (least significant) bit of a number always determines whether or not the number is odd or even– if bit 0 is 0, then the number is even, otherwise the number is odd.
- The largest positive number that can be represented in two's complement notation in an n-bit binary number is $2^{n-1} - 1$. For example, if $n = 8$, then the largest positive number is $01111111 = 2^7 - 1 = 127$.
- Similarly, the "most negative" number is -2^{n-1}, so if $n = 8$, then it is 10000000, which is $-2^7 = -128$. Note that the negative of the most negative number (in this case, 128) cannot be represented in this notation.

Addition and Subtraction of Signed Binary Numbers

The same addition algorithm that was used for unsigned binary numbers also works properly for two's complement numbers.

$$\begin{array}{rrcr} & 00000101 & = & 5 \\ + & 11110101 & = & \text{-}11 \\ \hline & 11111010 & = & \text{-}6 \end{array}$$

Subtraction is also done in a similar way: to subtract A from B, take the two's complement of A and then add this number to B.

The conditions for detecting overflow are different for signed and unsigned numbers, however. If we use algorithm C.3 to add two unsigned numbers, then if c is 1 when the addition terminates, this indicates that the result has an absolute value too large to fit the number of bits allowed. With signed numbers, however, c is not relevant, and an overflow occurs when the signs of both numbers being added are the same but the sign of the result is opposite. If the two numbers being added have opposite signs, however, then an overflow *cannot* occur.

For example, consider the sum of 1 and -1:

$$\begin{array}{rrcrl} & 00000001 & = & 1 & \\ + & 11111111 & = & \text{-}1 & \\ \hline & 00000000 & = & 0 & \textbf{Correct!} \end{array}$$

In this case, the addition will overflow, but it is not an error, since the result that we get (without considering the overflow) is exactly correct.

On the other hand, if we compute the sum of 127 and 1, then a serious error occurs:

$$\begin{array}{rrcrl} & 01111111 & = & 127 & \\ + & 00000001 & = & 1 & \\ \hline & 10000000 & = & \text{-}128 & \textbf{Uh-oh!} \end{array}$$

Therefore, we must be very careful when doing signed binary arithmetic that we take steps to detect bogus results. In general:

- If A and B are of the same sign, but $A + B$ is of the opposite sign, then an overflow or wraparound error has occurred.

- If A and B are of different signs, then $A + B$ will never overflow or wraparound.

Shifting Signed Binary Numbers

Another useful property of the two's complement notation is the ease with which numbers can be multiplied or divided by two. To multiply a number by two, simply shift the number "up" (to the left) by one bit, placing a 0 in the least significant bit. To divide a number in half, simply shift the the number "down" (to the right) by one bit (but do not change the sign bit).

Note that in the case of odd numbers, the effect of shifting to the right one bit is like dividing in half, rounded towards $-\infty$, so that 51 shifted to the right one bit becomes 25, while -51 shifted to the right one bit becomes -26.

	00000001	=	1
Double	00000010	=	2
Halve	00000000	=	0
	00110011	=	51
Double	01100110	=	102
Halve	00011001	=	25
	11001101	=	-51
Double	10011010	=	-102
Halve	11100110	=	-26

Hexadecimal Notation

Writing numbers in binary notation can soon get tedious, since even relatively small numbers require many binary digits to express. A more compact notation, called *hexadecimal* (base 16), is usually used to express large binary numbers. In hexadecimal, each digit represents four unsigned binary digits.

Another notation, which is not as common currently, is called *octal* and uses base eight to represent groups of three bits. Figure C.4 show examples of binary, decimal, octal, and hexadecimal numbers.

For example, the number $\texttt{200}_{10}$ can be written as $\texttt{11001000}_2$, $\texttt{C8}_{16}$, or $\texttt{310}_8$.

Figure C.4: Hexadecimal and Octal

Binary	0000	0001	0010	0011	0100	0101	0110	0111
Decimal	0	1	2	3	4	5	6	7
Hex	0	1	2	3	4	5	6	7
Octal	0	1	2	3	4	5	6	7

Binary	1000	1001	1010	1011	1100	1101	1110	1111
Decimal	8	9	10	11	12	13	14	15
Hex	8	9	A	B	C	D	E	F
Octal	10	11	12	13	14	15	16	17

C.2 Representing Characters

Just as sequences of bits can be used to represent numbers, they can also be used to represent the letters of the alphabet, as well as other characters.

Since all sequences of bits represent numbers, one way to think about representing characters by sequences of bits is to choose a number that corresponds to each character. The most popular correspondence currently is the ASCII character set. ASCII, which stands for the American Standard Code for Information Interchange, uses 7-bit integers to represent characters, using the correspondence shown in table C.5.

When the ASCII character set was chosen, some care was taken to organize the way that characters are represented in order to make them easy for a computer to manipulate. For example, all of the letters of the alphabet are arranged in order, so that sorting characters into alphabetical order is the same as sorting in numerical order. In addition, different classes of characters are arranged to have useful relations. For example, to convert the code for a lowercase letter to the code for the same letter in uppercase, simply set the 6th bit of the code to 0 (or subtract 32). ASCII is by no means the only character set to have similar useful properties, but it has emerged as the standard.

The ASCII character set does have some important limitations, however. One problem is that the character set only defines the representations of the characters used in written English. This causes problems with using ASCII to represent other written languages. In particular, there simply aren't enough bits to represent all the written characters of languages with a larger number of characters (such as Chinese or Japanese). Already new character sets which address these problems (and can be used to represent characters of many languages

Figure C.5: The ASCII Character Set

00 NUL	01 SOH	02 STX	03 ETX	04 EOT	05 ENQ	06 ACK	07 BEL
08 BS	09 HT	0A NL	0B VT	0C NP	0D CR	0E SO	0F SI
10 DLE	11 DC1	12 DC2	13 DC3	14 DC4	15 NAK	16 SYN	17 ETB
18 CAN	19 EM	1A SUB	1B ESC	1C FS	1D GS	1E RS	1F US
20 SP	21 !	22 "	23 #	24 $	25 %	26 &	27 '
28 (	29)	2A *	2B +	2C ,	2D -	2E .	2F /
30 0	31 1	32 2	33 3	34 4	35 5	36 6	37 7
38 8	39 9	3A :	3B ;	3C <	3D =	3E >	3F ?
40 @	41 A	42 B	43 C	44 D	45 E	46 F	47 G
48 H	49 I	4A J	4B K	4C L	4D M	4E N	4F O
50 P	51 Q	52 R	53 S	54 T	55 U	56 V	57 W
58 X	59 Y	5A Z	5B [	5C	5D]	5E ^	5F _
60 `	61 a	62 b	63 c	64 d	65 e	66 f	67 g
68 h	69 i	6A j	6B k	6C l	6D m	6E n	6F o
70 p	71 q	72 r	73 s	74 t	75 u	76 v	77 w
78 x	79 y	7A z	7B {	7C |	7D }	7E ~	7F DEL

side by side) are being proposed, and eventually there will unquestionably be a shift away from ASCII to a new multilanguage standard[1].

C.3 Representing Programs

Just as groups of bits can be used to represent numbers, they can also be used to represent instructions for a computer to perform. Unlike the two's complement notation for integers, which is a standard representation used by nearly all computers, the representation of instructions, and even the set of instructions, varies widely from one type of computer to another.

The MIPS architecture, which is the focus of later chapters in this document, uses a relatively simple and straightforward representation. Each instruction is exactly 32 bits in length, and consists of several bit fields, as depicted in figure C.6.

[1]This shift will break many, many existing programs. Converting all of these programs will keep many, many programmers busy for some time.

Figure C.6: MIPS R2000 Instruction Formats

	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits
Register	op	reg1	reg2	des	shift	funct
Immediate	op	reg1	reg2	16-bit constant		
Jump	op	26-bit constant				

The first six bits (reading from the left, or high-order bits) of each instruction are called the `op` field. The op field determines whether the instruction is a *register*, *immediate*, or *jump* instruction, and how the rest of the instruction should be interpreted. Depending on what the op is, parts of the rest of the instruction may represent the names of registers, constant memory addresses, 16-bit integers, or other additional qualifiers for the op.

If the op field is 0, then the instruction is a register instruction, which generally perform an arithmetic or logical operations. The *funct* field specifies the operation to perform, while the *reg1* and *reg2* represent the registers to use as operands, and the *des* field represents the register in which to store the result. For example, the 32-bit hexadecimal number `0x02918020` represents, in the MIPS instruction set, the operation of adding the contents of registers 20 and 17 and placing the result in register 16.

Field	op	reg1	reg2	des	shift	funct
Width	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits
Values	0	20	17	16	0	`add`
Binary	000000	10100	10001	10000	00000	100000

If the op field is not 0, then the instruction may be either an *immediate* or *jump* instruction, depending on the value of the op field.

C.4 Memory Organization

We've seen how sequences of binary digits can be used to represent numbers, characters, and instructions. In a computer, these binary digits are organized and manipulated in discrete groups, and these groups are said to be the *memory* of the computer.

C.4.1 Units of Memory

The smallest of these groups, on most computers, is called a *byte*. On nearly all currently popular computers a byte is composed of 8 bits.

The next largest unit of memory is usually composed of 16 bits. What this unit is called varies from computer to computer– on smaller machines, this is often called a *word*, while on newer architectures that can handle larger chunks of data, this is called a *halfword*.

The next largest unit of memory is usually composed of 32 bits. Once again, the name of this unit varies– on smaller machines, it is referred to as a *long*, while on newer and larger machines it is called a *word*.

Finally, on the newest machines, the computer also can handle data in groups of 64 bits. On a smaller machine, this is known as a *quadword*, while on a larger machine this is known as a *long*.

Historical Perspective

There have been architectures that have used nearly every imaginable word size– from 6-bit bytes to 9-bit bytes, and word sizes ranging from 12 bits to 48 bits. There are even a few architectures that have no fixed word size at all (such as the CM-2) or word sizes that can be specified by the operating system at runtime.

Over the years, however, most architectures have converged on 8-bit bytes and 32-bit longwords. An 8-bit byte is a good match for the ASCII character set (which has some popular extensions that require 8 bits), and a 32-bit word has been, at least until recently, large enough for most practical purposes.

C.4.2 Addresses and Pointers

Each unique byte[2] of the computer's memory is given a unique identifier, known as its *address*. The *address of* a piece of memory is often refered to as a *pointer to* that piece of memory– the two terms are synonymous, although there are many contexts where one is commonly used and the other is not.

The memory of the computer itself can often be thought of as a large array (or group of arrays) of bytes of memory. In this model, the address of each byte of memory is simply the index of the memory array location where that byte is stored.

[2]In some computers, the smallest distinct unit of memory is not a byte. For the sake of simplicity, however, this section assumes that the smallest distinct unit of memory on the computer in question is a byte.

C.4.3 Summary

In this chapter, we've seen how computers represent integers using groups of bits, and how basic arithmetic and other operations can be performed using this representation.

We've also seen how the integers or groups of bits can be used to represent several different kinds of data, including written characters (using the ASCII character codes), instructions for the computer to execute, and addresses or pointers, which can be used to reference other data.

There are also many other ways that information can be represented using groups of bits, including representations for rational numbers (usually by a representation called *floating point*), irrational numbers, graphics, arbitrary character sets, and so on. These topics, unfortunately, are beyond the scope of this chapter.

C.5 Exercises

1. Complete the following table:

Decimal	123				
Binary		01101100			
Octal			143		
Hex				3D	
ASCII					Z

2. Invent an algorithm for multiplying two unsigned binary numbers. You may find it easiest to start by thinking about multiplication of decimal numbers (there are other ways as well, but you should start on familiar ground).

3. (a) Invent an algorithm for dividing two unsigned binary numbers. You may find it easiest to start by thinking about long division of decimal numbers.

   (b) A user complains that the division algorithm you invented to solve the previous part of this problem is too slow. For her application, she would prefer an algorithm that gets an answer that is "reasonably close" to the right answer, but which may take considerably less time to compute. Invent an algorithm that has this property. Find the relationship between "reasonably close" and the speed of your algorithm.