CHAPTER III

Sorting and Searching Data

Few problems in computer science have been studied as much as sorting. Many good books are available that cover the subject in great depth. This chapter serves merely as an introduction, with an emphasis on practical applications in C.

Sorting

Five basic kinds of sorting algorithms are available to the programmer:

- Insertion sorts
- Exchange sorts
- Selection sorts
- Merge sorts
- Distribution sorts

An easy way to visualize how each sorting algorithm works is to think about how to sort a shuffled deck of cards lying on the table using each method. The cards are to be sorted by suit (clubs, diamonds, hearts, and spades) as well as by rank (2 through ace). You might have seen some of these algorithms in action at your last bridge game.

In an insertion sort, you pick up the cards one at a time, starting with the top card in the pile, and insert them into the correct position in your hand. When you have picked up all the cards, they are sorted.

In an exchange sort, you pick up the top two cards. If the one on the left belongs after the one on the right, you exchange the two cards' positions. Then you pick up the next card and perform the compare on the two rightmost cards and (if needed) exchange their positions. You repeat the process until all the cards are in your hand. If you didn't have to exchange any cards, the deck is sorted. Otherwise, you put down the deck and repeat the entire procedure until the deck is sorted.

In a selection sort, you search the deck for the lowest card and pick it up. You repeat the process until you are holding all the cards.

To perform a merge sort, you deal the deck into 52 piles of one card each. Because each pile is ordered (remember, there's only one card in it), if you merge adjacent piles, keeping cards in order, you will have 26 ordered piles of 2 cards each. You repeat so that you have 13 piles of 4 cards, then 7 piles (6 piles of 8 cards and 1 pile of 4 cards), until you have 1 pile of 52 cards.

In a distribution (or radix) sort, you deal the cards into 13 piles, placing each rank on its own pile. You then pick up all the piles in order and deal the cards into 4 piles, placing each suit on its own pile. You put the four piles together, and the deck is sorted.

There are several terms you should be aware of when examining sorting algorithms. The first is *natural.* A sort is said to be natural if it works faster (does less work) on data that is already sorted, and works slower (does more work) on data that is more mixed up. It is important to know whether a sort is natural if the data you're working with is already close to being sorted.

A sort is said to be *stable* if it preserves the ordering of data that are considered equal by the algorithm. Consider the following list:

Mary Jones
Mary Smith
Tom Jones
Susie Queue

If this list is sorted by last name using a stable sort, "Mary Jones" will remain before "Tom Jones" in the sorted list because they have the same last name. A stable sort can be used to sort data on a primary and secondary key, such as first and last names (in other words, sorted primarily by last name, but sorted by first name for entries with the same last name). This action is accomplished by sorting first on the secondary key, then on the primary key with a stable sort.

A sort that operates on data kept entirely in RAM is an *internal* sort. If a sort operates on data on disk, tape, or other secondary storage, it is called an *external* sort.

Searching

Searching algorithms have been studied nearly as much as sorting algorithms. The two are related in that many searching algorithms rely on the ordering of the data being searched. There are four basic kinds of searching algorithms:

- Sequential searching
- Comparison searching
- Radix searching
- Hashing

Each of these methods can be described using the same deck of cards example that was used for sorting.

In sequential searching, you go through the deck from top to bottom, looking at each card until you find the card you are looking for.

In comparison searching (also called binary searching), you start with an already sorted deck. You pick a card from the exact middle of the deck and compare it to the card you want. If it matches, you're done. Otherwise, if it's lower, you try the same search again in the first half of the deck. If it's higher, you try again in the second half of the deck.

In radix searching, you deal the deck into the 13 piles as described in radix *sorting*. To find a desired card, you choose the pile corresponding to the desired rank and search for the card you want in that pile using any search method. Or you could deal the deck into 4 piles based on suit as described in radix *sorting*. You could then pick the pile according to the desired suit and search there for the card you want.

In hashing, you make space for some number of piles on the table, and you choose a function that maps cards into a particular pile based on rank and suit (called a *hash* function). You then deal all the cards into the piles, using the hash function to decide where to put each card. To find a desired card, you use the hash function to find out which pile the desired card should be in. Then you search for the card in that pile.

For instance, you might make 16 piles and pick a hash function like `pile = rank + suit`. If `rank` is a card's rank treated as a number (ace = 1, 2 = 2, all the way up to king = 13), and `suit` is a card's suit treated as a number (clubs = 0, diamonds = 1, hearts = 2, spades = 3), then for each card you can compute a `pile` number that will be from 1 to 16, indicating which pile the card belongs in. This technique sounds crazy, but it's a very powerful searching method. All sorts of programs, from compression programs (such as Stacker) to disk caching programs (such as SmartDrive) use hashing to speed up searches for data.

Performance of Sorting or Searching

One of the chief concerns in searching and sorting is speed. Often, this concern is misguided, because the sort or search takes negligible time compared to the rest of the program. For most sorting and searching applications, you should use the easiest method available (see FAQs III.1 and III.4). If you later find that the program is too slow because of the searching or sorting algorithm used, you can substitute another method easily. By starting with a simple method, you haven't invested much time and effort on code that has to be replaced.

One measure of the speed of a sorting or searching algorithm is the number of operations that must be performed in the best, average, and worst cases (sometimes called the algorithm's *complexity*). This is described by a mathematical expression based on the number of elements in the data set to be sorted or searched. This expression shows how fast the execution time of the algorithm grows as the number of elements increases. It does not tell how fast the algorithm is for a given size data set, because that rate depends on the exact implementation and hardware used to run the program.

The fastest algorithm is one whose complexity is *O(1)* (which is read as "order 1"). This means that the number of operations is not related to the number of elements in the data set. Another common complexity for algorithms is *O(N)* (*N* is commonly used to refer to the number of elements in the data set). This means that the number of operations is directly related to the number of elements in the data set. An algorithm with complexity *O(log N)* is somewhere between the two in speed. The *O(log N)* means that the number of operations is related to the logarithm of the number of elements in the data set.

NOTE

If you're unfamiliar with the term, you can think of a *log N* as the number of digits needed to write the number *N*. Thus, the *log 34* is 2, and the *log 900* is 3 (actually, *log 10* is 2 and *log 100* is 3—*log 34* is a number between 2 and 3).

If you're still with me, I'll add another concept. A logarithm has a particular base. The logarithms described in the preceding paragraph are base 10 logarithms (written as *log10 N*), meaning that if *N* gets 10 times as big, *log N* gets bigger by 1. The base can be any number. If you are comparing two algorithms, both of which have complexity *O(log N)*, the one with the larger base would be faster. No matter what the base is, *log N* is always a smaller number than *N*.

An algorithm with complexity *O(N log N)* (*N* times *log N*) is slower than one of complexity *O(N)*, and an algorithm of complexity $O(N^2)$ is slower still. So why don't they just come up with one algorithm with the lowest complexity number and use only that one? Because the complexity number only describes how the program will slow down as *N* gets larger.

The complexity does not indicate which algorithm is faster for any particular value of *N*. That depends on many factors, including the type of data in the set, the language the algorithm is written in, and the machine and compiler used. What the complexity number does communicate is that as *N* gets bigger, there will be a point at which an algorithm with a lower order complexity will be faster than one of a higher order complexity.

Table 3.1 shows the complexity of all the algorithms listed in this chapter. The best and worst cases are given for the sorting routines. Depending on the original order of the data, the performance of these algorithms will vary between best and worst case. The average case is for randomly ordered data. The average case complexity for searching algorithms is given. The best case for all searching algorithms (which is if the data happens to be in the first place searched) is obviously *O(1)*. The worst case (which is if the data being searching for doesn't exist) is generally the same as the average case.

Table 3.1. The relative complexity of all the algorithms presented in this chapter.

Algorithm	*Best*	*Average*	*Worst*
Quick sort	*O(N log N)*	*O(N log N)*	$O(N^2)$
Merge sort	*O(N)*	*O(N log N)*	*O(N log N)*
Radix sort	*O(N)*	*O(N)*	*O(N)*
Linear search		*O(N)*	
Binary search		*O(log N)*	
Hashing		*O(N/M)**	
Digital trie		*O(1)***	

* *M* is the size of hash table

** Actually, equivalent to a hash table with 2^{32} entries

To illustrate the difference between the complexity of an algorithm and its actual running time, Table 3.2 shows execution time for all the sample programs listed in this chapter. Each program was compiled with the GNU C Compiler (gcc) Version 2.6.0 under the Linux operating system on a 90 MHz Pentium computer. Different computer systems should provide execution times that are proportional to these times.

Table 3.2. The execution times of all the programs presented in this chapter.

Program	*Algorithm*	*2000*	*4000*	*6000*	*8000*	*10000*
3_1	`qsort()`	0.02	0.05	0.07	0.11	0.13
3_2a	quick sort	0.02	0.06	0.13	0.18	0.20
3_2b	merge sort	0.03	0.08	0.14	0.18	0.26
3_2c	radix sort	0.07	0.15	0.23	0.30	0.39
3_4	`bsearch()`	0.37	0.39	0.39	0.40	0.41
3_5	binary search	0.32	0.34	0.34	0.36	0.36
3_6	linear search	9.67	20.68	28.71	36.31	45.51
3_7	trie search	0.27	0.28	0.29	0.29	0.30
3_8	hash search	0.25	0.26	0.28	0.29	0.28

NOTE

All times are in seconds. Times are normalized, by counting only the time for the program to perform the sort or search.

The 2000–10000 columns indicate the number of elements in the data set to be sorted or searched. Data elements were words chosen at random from the file /usr/man/man1/gcc.1 (documentation for the GNU C compiler).

For the search algorithms, the data searched for were words chosen at random from the file /usr/man/man1/g++.1 (documentation for the GNU C++ compiler).

`qsort()` and `bsearch()` are standard library implementations of quick sort and binary search, respectively. The rest of the programs are developed from scratch in this chapter.

This information should give you a taste of what issues are involved in deciding which algorithm is appropriate for sorting and searching in different situations. The book *The Art of Computer Programming, Volume 3, Sorting and Searching*, by Donald E. Knuth, is entirely devoted to algorithms for sorting and searching, with much more information on complexity and complexity theory as well as many more algorithms than are described here.

Some Code to Get Started With

This chapter includes several code examples that are complete enough to actually compile and run. To avoid duplicating the code that is common to several examples, the code is shown at the end of this chapter.

III.1: What is the easiest sorting method to use?

Answer:

The answer is the standard library function `qsort()`. It's the easiest sort by far for several reasons:

It is already written.

It is already debugged.

It has been optimized as much as possible (usually).

The algorithm used by `qsort()` is generally the quick sort algorithm, developed by C. A. R. Hoare in 1962. Here is the prototype for `qsort()`:

```
void qsort(void *buf, size_t num, size_t size,
          int (*comp)(const void *ele1, const void *ele2));
```

The `qsort()` function takes a pointer to an array of user-defined data (`buf`). The array has `num` elements in it, and each element is `size` bytes long. Decisions about sort order are made by calling `comp`, which is a pointer to a function that compares two elements of `buf` and returns a value that is less than, equal to, or greater than 0 according to whether the `ele1` is less than, equal to, or greater than `ele2`.

For instance, say you want to sort an array of strings in alphabetical order. The array is terminated by a `NULL` pointer. Listing III.1 shows the function `sortStrings()`, which sorts a `NULL`-terminated array of character strings using the `qsort()` function. You can compile this example into a working program using the code found at the end of this chapter.

Listing III.1. An example of using `qsort()`.

```
 1: #include       <stdlib.h>
 2:
 3: /*
 4:  * This routine is used only by sortStrings(), to provide a
 5:  * string comparison function to pass to qsort().
 6:  */
 7: static int comp(const void *ele1, const void *ele2)
 8: {
 9:         return strcmp(*(const char **) ele1,
10:                 *(const char **) ele2);
11: }
12:
13: /* Sort strings using the library function qsort() */
14: void sortStrings(const char *array[])
15: {
16:         /* First, determine the length of the array */
17:         int     num;
18:
```

```
19:         for (num = 0; array[num]; num++)
20:                 ;
21:         qsort(array, num, sizeof(*array), comp);
22: }
```

The `for` loop on lines 19 and 20 simply counts the number of elements in the array so that the count can be passed to `qsort()`. The only "tricky" part about this code is the `comp()` function. Its sole purpose is to bridge the gap between the types that `qsort()` passes to it (`const void *`) and the types expected by `strcmp()` (`const char *`). Because `qsort()` works with pointers to elements, and the elements are themselves pointers, the correct type to cast `ele1` and `ele2` to is `const char **`. The result of the cast is then dereferenced (by putting the `*` in front of it) to get the `const char *` type that `strcmp()` expects.

Given that `qsort()` exists, why would a C programmer ever write another sort program? There are several reasons. First, there are pathological cases in which `qsort()` performs very slowly and other algorithms perform better. Second, some overhead is associated with `qsort()` because it is general purpose. For instance, each comparison involves an indirect function call through the function pointer provided by the user. Also, because the size of an element is a runtime parameter, the code to move elements in the array isn't optimized for a single size of element. If these performance considerations are important, writing a sort routine might be worth it.

Besides the drawbacks mentioned, the `qsort()` implementation assumes that all the data is in one array. This might be inconvenient or impossible given the size or nature of the data. Lastly, `qsort()` implementations are usually not "stable" sorts.

Cross Reference:

III.2: What is the quickest sorting method to use?

III.3: How can I sort things that are too large to bring into memory?

III.7: How can I sort a linked list?

VII.1: What is indirection?

VII.2: How many levels of pointers can you have?

VII.5: What is a `void` pointer?

VII.6: When is a `void` pointer used?

III.2: What is the quickest sorting method to use?

Answer:

The answer depends on what you mean by quickest. For most sorting problems, it just doesn't matter how quick the sort is because it is done infrequently or other operations take significantly more time anyway. Even in cases in which sorting speed is of the essence, there is no one answer. It depends on not only the size and nature of the data, but also the likely order. No algorithm is best in all cases.

There are three sorting methods in this author's "toolbox" that are all very fast and that are useful in different situations. Those methods are quick sort, merge sort, and radix sort.

The Quick Sort

The quick sort algorithm is of the "divide and conquer" type. That means it works by reducing a sorting problem into several easier sorting problems and solving each of them. A "dividing" value is chosen from the input data, and the data is partitioned into three sets: elements that belong before the dividing value, the value itself, and elements that come after the dividing value. The partitioning is performed by exchanging elements that are in the first set but belong in the third with elements that are in the third set but belong in the first. Elements that are equal to the dividing element can be put in any of the three sets—the algorithm will still work properly.

After the three sets are formed, the middle set (the dividing element itself) is already sorted, so quick sort is applied to the first and third sets, recursively. At some point, the set being sorting becomes too small for quick sort. Obviously, a set of two or fewer elements cannot be divided into three sets. At this point, some other sorting method is used. The cutoff point at which a different method of sorting is applied is up to the person implementing the sort. This cutoff point can dramatically affect the efficiency of the sort, because there are methods that are faster than quick sort for relatively small sets of data.

The string sorting example (from FAQ III.1) will be rewritten using a quick sort. Excuse the preprocessor trickery, but the goal is to make the code readable and fast. Listing III.2a shows `myQsort()`, an implementation of the quick sort algorithm from scratch. You can compile this example into a working program using the code at the end of this chapter.

The function `myQsort()` sorts an array of strings into ascending order. First it checks for the simplest cases. On line 17 it checks for the case of zero or one element in the array, in which case it can return—the array is already sorted. Line 19 checks for the case of an array of two elements, because this is too small an array to be handled by the rest of the function. If there are two elements, either the array is sorted or the two elements are exchanged to make the array sorted.

Line 28 selects the middle element of the array as the one to use to partition the data. It moves that element to the beginning of the array and begins partitioning the data into two sets. Lines 37–39 find the first element in the array that belongs in the second set, and lines 45–47 find the last element in the array that belongs in the first set.

Line 49 checks whether the first element that belongs in the second set is after the last element that belongs in the first set. If this is the case, *all* the elements in the first set come before the elements in the second set, so the data are partitioned. Otherwise, the algorithm swaps the two elements so that they will be in the proper set, and then continues.

After the array has been properly partitioned into two sets, line 55 puts the middle element back into its proper place between the two sets, which turns out to be its correct position in the sorted array. Lines 57 and 58 sort each of the two sets by calling `myQsort()` recursively. When each set is sorted, the entire array is sorted.

Listing III.2a. An implementation of quick sort that doesn't use the `qsort()` function.

```
1: #include        <stdlib.h>
2:
3: #define exchange(A, B, T)       ((T) = (A), (A) = (B), \
4:                                                  (B) = (T))
5:
```

```
 6: /* Sorts an array of strings using quick sort algorithm */
 7: static void myQsort(const char *array[], size_t num)
 8: {
 9:         const char      *temp;
10:         size_t  i, j;
11:
12:         /*
13:          * Check the simple cases first:
14:          * If fewer than 2 elements, already sorted
15:          * If exactly 2 elements, just swap them (if needed).
16:          */
17:         if (num < 2)
18:                 return;
19:         else if (num == 2)
20:         {
21:                 if (strcmp(array[0], array[1]) > 0)
22:                         exchange(array[0], array[1], temp);
23:         }
24:         /*
25:          * Partition the array using the middle (num / 2)
26:          * element as the dividing element.
27:          */
28:         exchange(array[0], array[num / 2], temp);
29:         i = 1;
30:         j = num;
31:         for ( ; ; )
32:         {
33:                 /*
34:                  * Sweep forward until an element is found that
35:                  * belongs in the second partition.
36:                  */
37:                 while (i < j && strcmp(array[i], array[0])
38:                                                         <= 0)
39:                         i++;
40:                 /*
41:                  * Then sweep backward until an element
42:                  * is found that belongs in the first
43:                  * partition.
44:                  */
45:                 while (i < j && strcmp(array[j - 1], array[0])
46:                                                         >= 0)
47:                         j--;
48:                 /* If no out-of-place elements, you're done */
49:                 if (i >= j)
50:                         break;
51:                 /* Else, swap the two out-of-place elements */
52:                 exchange(array[i], array[j - 1], temp);
53:         }
54:         /* Restore dividing element */
55:         exchange(array[0], array[i - 1], temp);
56:         /* Now apply quick sort to each partition */
57:         myQsort(array, i - 1);
58:         myQsort(array + i, num - i);
59: }
60:
61: /* Sort strings using your own implementation of quick sort */
62: void sortStrings(const char *array[])
```

continues

Listing III.2a. continued

```
63: {
64:         /* First, determine the length of the array */
65:         int     num;
66:
67:         for (num = 0; array[num]; num++)
68:                 ;
69:         myQsort((void *) array, num);
70: }
```

The Merge Sort

The merge sort is a "divide and conquer" sort as well. It works by considering the data to be sorted as a sequence of already-sorted lists (in the worst case, each list is one element long). Adjacent sorted lists are merged into larger sorted lists until there is a single sorted list containing all the elements. The merge sort is good at sorting lists and other data structures that are not in arrays, and it can be used to sort things that don't fit into memory. It also can be implemented as a stable sort. The merge sort was suggested by John von Neumann in 1945!

Listing III.2b shows an implementation of the merge sort algorithm. To make things more interesting, the strings will be put into a linked list structure rather than an array. In fact, the algorithm works better on data that is organized as lists, because elements in an array cannot be merged in place (some extra storage is required). You can compile this example into a working program using the code at the end of this chapter. The code for (and a description of) the `list_t` type and the functions that operate on `list_t`s are also at the end of this chapter.

There are four functions that together implement merge sort. The function `split()` takes a list of strings and turns it into a list of lists of strings, in which each list of strings is sorted. For instance, if the original list was ("the" "quick" "brown" "fox"), `split()` would return a list of three lists—("the"), ("quick"), and ("brown" "fox")—because the strings "brown" and "fox" are already in the correct order. The algorithm would work just as well if `split()` made lists of one element each, but splitting the list into already-sorted chunks makes the algorithm natural by reducing the amount of work left to do if the list is nearly sorted already (see the introduction to this chapter for a definition of natural sorts). In the listing, the loop on lines 14–24 keeps processing as long as there are elements on the input list. Each time through the loop, line 16 makes a new list, and the loop on lines 17–22 keeps moving elements from the input list onto this list as long as they are in the correct order. When the loop runs out of elements on the input list or encounters two elements out of order, line 23 appends the current list to the output list of lists.

The function `merge()` takes two lists that are already sorted and merges them into a single sorted list. The loop on lines 37–45 executes as long as there is something on both lists. The `if` statement on line 40 selects the smaller first element of the two lists and moves it to the output list. When one of the lists becomes empty, all the elements of the other list must be appended to the output list. Lines 46 and 47 concatenate the output list with the empty list and the non-empty list to complete the merge.

The function `mergePairs()` takes a list of lists of strings and calls `merge()` on each pair of lists of strings, replacing the original pair with the single merged list. The loop on lines 61–77 executes as long as there is something in the input list. The `if` statement on line 63 checks whether there are at least two lists of strings on the input list. If not, line 76 appends the odd list to the output list. If so, lines 65 and 66 remove the two

lists, which are merged on lines 68 and 69. The new list is appended to the output list on line 72, and all the intermediate list nodes that were allocated are freed on lines 70, 71, and 73. Lines 72 and 73 remove the two lists that were merged from the input list.

The last function is `sortStrings()`, which performs the merge sort on an array of strings. Lines 88 and 89 put the strings into a list. Line 90 calls `split()` to break up the original list of strings into a list of lists of strings. The loop on lines 91 and 92 calls `mergePairs()` until there is only one list of strings on the list of lists of strings. Line 93 checks to ensure that the list isn't empty (which is the case if the array has 0 elements in it to begin with) before removing the sorted list from the list of lists. Finally, lines 95 and 96 put the sorted strings back into the array. Note that `sortStrings()` does not free all the memory if allocated.

Listing III.2b. An implementation of a merge sort.

```
 1: #include        <stdlib.h>
 2: #include        "list.h"
 3:
 4: /*
 5:  * Splits a list of strings into a list of lists of strings
 6:  * in which each list of strings is sorted.
 7:  */
 8: static list_t split(list_t in)
 9: {
10:         list_t  out;
11:         list_t  *curr;
12:         out.head = out.tail = NULL;
13:
14:         while (in.head)
15:         {
16:                 curr = newList();
17:                 do
18:                 {
19:                         appendNode(curr, removeHead(&in));
20:                 }
21:                 while (in.head && strcmp(curr->tail->u.str,
22:                                 in.head->u.str) <= 0);
23:                 appendNode(&out, newNode(curr));
24:         }
25:         return out;
26: }
27:
28: /*
29:  * Merge two sorted lists into a third sorted list,
30:  * which is then returned.
31:  */
32: static list_t merge(list_t first, list_t second)
33: {
34:         list_t  out;
35:         out.head = out.tail = NULL;
36:
37:         while (first.head && second.head)
38:         {
39:                 listnode_t      *temp;
40:                 if (strcmp(first.head->u.str,
41:                                 second.head->u.str) <= 0)
```

continues

Listing III.2b. continued

```
42:                         appendNode(&out, removeHead(&first));
43:                 else
44:                         appendNode(&out, removeHead(&second));
45:         }
46:         concatList(&out, &first);
47:         concatList(&out, &second);
48:         return out;
49: }
50:
51: /*
52:  * Takes a list of lists of strings and merges each pair of
53:  * lists into a single list. The resulting list has 1/2 as
54:  * many lists as the original.
55:  */
56: static list_t mergePairs(list_t in)
57: {
58:         list_t  out;
59:         out.head = out.tail = NULL;
60:
61:         while (in.head)
62:         {
63:                 if (in.head->next)
64:                 {
65:                         list_t  *first = in.head->u.list;
66:                         list_t  *second =
67:                                         in.head->next->u.list;
68:                         in.head->u.list = copyOf(merge(*first,
69:                                                 *second));
70:                         free(first);
71:                         free(second);
72:                         appendNode(&out, removeHead(&in));
73:                         free(removeHead(&in));
74:                 }
75:                 else
76:                         appendNode(&out, removeHead(&in));
77:         }
78:         return out;
79: }
80:
81: /* Sort strings using merge sort */
82: void sortStrings(const char *array[])
83: {
84:         int     i;
85:         list_t  out;
86:         out.head = out.tail = NULL;
87:
88:         for (i = 0; array[i]; i++)
89:                 appendNode(&out, newNode((void *) array[i]));
90:         out = split(out);
91:         while (out.head != out.tail)
92:                 out = mergePairs(out);
93:         if (out.head)
94:                 out = *out.head->u.list;
95:         for (i = 0; array[i]; i++)
96:                 array[i] = removeHead(&out)->u.str;
97: }
```

The Radix Sort

The radix sort shown in Listing III.2c takes a list of integers and puts each element on a smaller list, depending on the value of its least significant byte. Then the small lists are concatenated, and the process is repeated for each more significant byte until the list is sorted. The radix sort is simpler to implement on fixed-length data such as `int`s, but it is illustrated here using strings. You can compile this example into a working program using the code at the end of this chapter.

Two functions perform the radix sort. The function `radixSort()` performs one pass through the data, performing a partial sort. Line 12 ensures that all the lists in `table` are empty. The loop on lines 13–24 executes as long as there is something on the input list. Lines 15–22 select which position in the table to put the next string on, based on the value of the character in the string specified by `whichByte`. If the string has fewer characters than `whichByte` calls for, the position is 0 (which ensures that the string "an" comes before the string "and"). Finally, lines 25 and 26 concatenate all the elements of `table` into one big list in `table[0]`.

The function `sortStrings()` sorts an array of strings by calling `radixSort()` several times to perform partial sorts. Lines 39–46 create the original list of strings, keeping track of the length of the longest string (because that's how many times it needs to call `radixSort()`). Lines 47 and 48 call `radixSort()` for each byte in the longest string in the list. Finally, lines 49 and 50 put all the strings in the sorted list back into the array. Note that `sortStrings()` doesn't free all the memory it allocates.

Listing III.2c. An implementation of a radix sort.

```
 1: #include        <stdlib.h>
 2: #include        <limits.h>
 3: #include        <memory.h>
 4: #include        "list.h"
 5:
 6: /* Partially sort list using radix sort */
 7: static list_t radixSort(list_t in, int whichByte)
 8: {
 9:         int     i;
10:         list_t  table[UCHAR_MAX + 1];
11:
12:         memset(table, 0, sizeof(table));
13:         while (in.head)
14:         {
15:                 int     len = strlen(in.head->u.str);
16:                 int     pos;
17:
18:                 if (len > whichByte)
19:                         pos = (unsigned char)
20:                                 in.head->u.str[whichByte];
21:                 else
22:                         pos = 0;
23:                 appendNode(&table[pos], removeHead(&in));
24:         }
25:         for (i = 1; i < UCHAR_MAX + 1; i++)
26:                 concatList(&table[0], &table[i]);
27:         return table[0];
28: }
29:
```

continues

Listing III.2c. continued

```
30: /* Sort strings using radix sort */
31: void sortStrings(const char *array[])
32: {
33:         int     i;
34:         int     len;
35:         int     maxLen = 0;
36:         list_t  list;
37:
38:         list.head = list.tail = NULL;
39:         for (i = 0; array[i]; i++)
40:         {
41:                 appendNode(&list,
42:                         newNode((void *) array[i]));
43:                 len = strlen(array[i]);
44:                 if (len > maxLen)
45:                         maxLen = len;
46:         }
47:         for (i = maxLen - 1; i >= 0; i--)
48:                 list = radixSort(list, i);
49:         for (i = 0; array[i]; i++)
50:                 array[i] = removeHead(&list)->u.str;
51: }
```

Cross Reference:

III.1: What is the easiest sorting method to use?

III.3: How can I sort things that are too large to bring into memory?

III.7: How can I sort a linked list?

III.3: How can I sort things that are too large to bring into memory?

Answer:

A sorting program that sorts items that are on secondary storage (disk or tape) rather than primary storage (memory) is called an *external* sort. Exactly how to sort large data depends on what is meant by "too large to fit in memory." If the items to be sorted are themselves too large to fit in memory (such as images), but there aren't many items, you can keep in memory only the sort key and a value indicating the data's location on disk. After the key/value pairs are sorted, the data is rearranged on disk into the correct order.

If "too large to fit in memory" means that there are too many items to fit into memory at one time, the data can be sorted in groups that will fit into memory, and then the resulting files can be merged. A sort such as a radix sort can also be used as an external sort, by making each bucket in the sort a file.

Even the quick sort can be an external sort. The data can be partitioned by writing it to two smaller files. When the partitions are small enough to fit, they are sorted in memory and concatenated to form the sorted file.

The example in Listing III.3 is an external sort. It sorts data in groups of 10,000 strings and writes them to files, which are then merged. If you compare this listing to the listing of the merge sort (Listing III.2b), you will notice many similarities.

Any of the four sort programs introduced so far in this chapter can be used as the in-memory sort algorithm (the makefile given at the end of the chapter specifies using `qsort()` as shown in Listing III.1). The functions `myfgets()` and `myfputs()` simply handle inserting and removing the newline (`'\n'`) characters at the ends of lines. The `openFile()` function handles error conditions during the opening of files, and `fileName()` generates temporary filenames.

The function `split()` reads in up to 10,000 lines from the input file on lines 69–74, sorts them in memory on line 76, and writes them to a temporary file on lines 77–80. The function `merge()` takes two files that are already sorted and merges them into a third file in exactly the same way that the `merge()` routine in Listing III.2b merged two lists. The function `mergePairs()` goes through all the temporary files and calls `merge()` to combine pairs of files into single files, just as `mergePairs()` in Listing III.2b combines lists. Finally, `main()` invokes `split()` on the original file, then calls `mergePairs()` until all the files are combined into one big file. It then replaces the original unsorted file with the new, sorted file.

Listing III.3. An example of an external sorting algorithm.

```
 1: #include        <stdlib.h>
 2: #include        <string.h>
 3: #include        <stdio.h>
 4: #include        <stdio.h>
 5:
 6: #define LINES_PER_FILE  10000
 7:
 8: /* Just like fgets(), but removes trailing '\n'. */
 9: char*
10: myfgets(char *buf, size_t size, FILE *fp)
11: {
12:         char    *s = fgets(buf, size, fp);
13:         if (s)
14:                 s[strlen(s) - 1] = '\0';
15:         return s;
16: }
17:
18: /* Just like fputs(), but adds trailing '\n'. */
19: void
20: myfputs(char *s, FILE *fp)
21: {
22:         int     n = strlen(s);
23:         s[n] = '\n';
24:         fwrite(s, 1, n + 1, fp);
25:         s[n] = '\0';
26: }
27:
28: /* Just like fopen(), but prints message and dies if error. */
29: FILE*
30: openFile(const char *name, const char *mode)
31: {
32:         FILE    *fp = fopen(name, mode);
33:
34:         if (fp == NULL)
```

continues

Listing III.3. continued

```
35:         {
36:                 perror(name);
37:                 exit(1);
38:         }
39:         return fp;
40: }
41:
42: /* Takes a number and generates a filename from it. */
43: const char*
44: fileName(int n)
45: {
46:         static char     name[16];
47:
48:         sprintf(name, "temp%d", n);
49:         return name;
50: }
51:
52: /*
53:  * Splits input file into sorted files with no more
54:  * than LINES_PER_FILE lines each.
55:  */
56: int
57: split(FILE *infp)
58: {
59:         int     nfiles = 0;
60:         int     line;
61:
62:         for (line = LINES_PER_FILE; line == LINES_PER_FILE; )
63:         {
64:                 char    *array[LINES_PER_FILE + 1];
65:                 char    buf[1024];
66:                 int     i;
67:                 FILE    *fp;
68:
69:                 for (line = 0; line < LINES_PER_FILE; line++)
70:                 {
71:                         if (!myfgets(buf, sizeof(buf), infp))
72:                                 break;
72:                         array[line] = strdup(buf);
74:                 }
75:                 array[line] = NULL;
76:                 sortStrings(array);
77:                 fp = openFile(fileName(nfiles++), "w");
78:                 for (i = 0; i < line; i++)
79:                         myfputs(array[i], fp);
80:                 fclose(fp);
81:         }
82:         return nfiles;
83: }
84:
85: /*
86:  * Merges two sorted input files into
87:  * one sorted output file.
88:  */
89: void
90: merge(FILE *outfp, FILE *fp1, FILE *fp2)
```

```
 91: {
 92:         char    buf1[1024];
 93:         char    buf2[1024];
 94:         char    *first;
 95:         char    *second;
 96:
 97:         first = myfgets(buf1, sizeof(buf1), fp1);
 98:         second = myfgets(buf2, sizeof(buf2), fp2);
 99:         while (first && second)
100:         {
101:                 if (strcmp(first, second) > 0)
102:                 {
103:                         myfputs(second, outfp);
104:                         second = myfgets(buf2, sizeof(buf2),
105:                                                         fp2);
106:                 }
107:                 else
108:                 {
109:                         myfputs(first, outfp);
110:                         first = myfgets(buf1, sizeof(buf1),
111:                                                         fp1);
112:                 }
113:         }
114:         while (first)
115:         {
116:                 myfputs(first, outfp);
117:                 first = myfgets(buf1, sizeof(buf1), fp1);
118:         }
119:         while (second)
120:         {
121:                 myfputs(second, outfp);
122:                 second = myfgets(buf2, sizeof(buf2), fp2);
123:         }
124: }
125:
126: /*
127:  * Takes nfiles files and merges pairs of them.
128:  * Returns new number of files.
129:  */
130: int
131: mergePairs(int nfiles)
132: {
133:         int     i;
134:         int     out = 0;
135:
136:         for (i = 0; i < nfiles - 1; i += 2)
137:         {
138:                 FILE    *temp;
139:                 FILE    *fp1;
140:                 FILE    *fp2;
141:                 const char      *first;
142:                 const char      *second;
143:
144:                 temp = openFile("temp", "w");
145:                 fp1 = openFile(fileName(i), "r");
146:                 fp2 = openFile(fileName(i + 1), "r");
147:                 merge(temp, fp1, fp2);
148:                 fclose(fp1);
```

continues

Listing III.3. continued

```
149:                    fclose(fp2);
150:                    fclose(temp);
151:                    unlink(fileName(i));
152:                    unlink(fileName(i + 1));
153:                    rename("temp", fileName(out++));
154:            }
155:            if (i < nfiles)
156:            {
157:                    char    *tmp = strdup(fileName(i));
158:                    rename(tmp, fileName(out++));
159:                    free(tmp);
160:            }
161:            return out;
162: }
163:
164: int
165: main(int argc, char **argv)
166: {
167:            char    buf2[1024];
168:            int     nfiles;
169:            int     line;
170:            int     in;
171:            int     out;
172:            FILE    *infp;
173:
174:            if (argc != 2)
175:            {
176:                    fprintf(stderr, "usage: %s file\n", argv[0]);
177:                    exit(1);
178:            }
179:            infp = openFile(argv[1], "r");
180:            nfiles = split(infp);
181:            fclose(infp);
182:            while (nfiles > 1)
183:                    nfiles = mergePairs(nfiles);
184:            rename(fileName(0), argv[1]);
185:            return 0;
186: }
```

Cross Reference:

III.1: What is the easiest sorting method to use?

III.2: What is the quickest sorting method to use?

III.7: How can I sort a linked list?

III.4: What is the easiest searching method to use?

Answer:

Just as `qsort()` was the easiest sorting method, because it is part of the standard library, `bsearch()` is the easiest searching method to use.

Following is the prototype for `bsearch()`:

```
void *bsearch(const void *key, const void *buf, size_t num, size_t size,
              int (*comp)(const void *, const void *));
```

The `bsearch()` function performs a binary search on an array of sorted data elements. A binary search is another "divide and conquer" algorithm. The key is compared with the middle element of the array. If it is equal, the search is done. If it is less than the middle element, the item searched for must be in the first half of the array, so a binary search is performed on just the first half of the array. If the key is greater than the middle element, the item searched for must be in the second half of the array, so a binary search is performed on just the second half of the array. Listing III.4a shows a simple function that calls the `bsearch()` function. This listing borrows the function `comp()` from Listing III.1, which used `qsort()`. Listing III.4b shows a binary search, performed without calling `bsearch()`, for a string in a sorted array of strings. You can make both examples into working programs by combining them with code at the end of this chapter.

Listing III.4a. An example of how to use `bsearch()`.

```
 1: #include        <stdlib.h>
 2:
 3: static int comp(const void *ele1, const void *ele2)
 4: {
 5:         return strcmp(*(const char **) ele1,
 6:                 *(const char **) ele2);
 7: }
 8:
 9: const char *search(const char *key, const char **array,
10:                                                 size_t num)
11: {
12:         char    **p = bsearch(&key, array, num,
13:                                         sizeof(*array), comp);
14:         return p ? *p : NULL;
15: }
```

Listing III.4b. An implementation of a binary search.

```
 1: #include        <stdlib.h>
 2:
 3: const char *search(const char *key, const char **array,
 4:                                                 size_t num)
 5: {
 6:         int     low = 0;
 7:         int     high = num - 1;
 8:
 9:         while (low <= high)
10:         {
11:                 int     mid = (low + high) / 2;
12:                 int     n = strcmp(key, array[mid]);
13:
14:                 if (n < 0)
15:                         high = mid - 1;
16:                 else if (n > 0)
17:                         low = mid + 1;
```

continues

Listing III.4b. continued

```
18:                 else
19:                         return array[mid];
20:         }
21:         return 0;
22: }
```

Another simple searching method is a linear search. A linear search is not as fast as `bsearch()` for searching among a large number of items, but it is adequate for many purposes. A linear search might be the only method available, if the data isn't sorted or can't be accessed randomly. A linear search starts at the beginning and sequentially compares the key to each element in the data set. Listing III.4c shows a linear search. As with all the examples in this chapter, you can make it into a working program by combining it with code at the end of the chapter.

Listing III.4c. An implementation of linear searching.

```
 1: #include        <stdlib.h>
 2:
 3: const char *search(const char *key, const char **array,
 4:                                                  size_t num)
 5: {
 6:         int     i;
 7:
 8:         for (i = 0; i < num; i++)
 9:         {
10:                 if (strcmp(key, array[i]) == 0)
11:                         return array[i];
12:         }
13:         return 0;
14: }
```

Cross Reference:

III.5: What is the quickest searching method to use?

III.6: What is hashing?

III.8: How can I search for data in a linked list?

III.5: What is the quickest searching method to use?

Answer:

A binary search, such as `bsearch()` performs, is much faster than a linear search. A hashing algorithm can provide even faster searching. One particularly interesting and fast method for searching is to keep the data in a "digital trie." A digital trie offers the prospect of being able to search for an item in essentially a constant amount of time, independent of how many items are in the data set.

A digital trie combines aspects of binary searching, radix searching, and hashing. The term "digital trie" refers to the data structure used to hold the items to be searched. It is a multilevel data structure that branches *N* ways at each level (in the example that follows, each level branches from 0 to 16 ways). The subject of treelike data structures and searching is too broad to describe fully here, but a good book on data structures or algorithms can teach you the concepts involved.

Listing III.5 shows a program implementing digital trie searching. You can combine this example with code at the end of the chapter to produce a working program. The concept is not too hard. Suppose that you use a hash function that maps to a full 32-bit integer. The hash value can also be considered to be a concatenation of eight 4-bit hash values. You can use the first 4-bit hash value to index into a 16-entry hash table.

Naturally, there will be many collisions, with only 16 entries. Collisions are resolved by having the table entry point to a second 16-entry hash table, in which the next 4-bit hash value is used as an index.

The tree of hash tables can be up to eight levels deep, after which you run out of hash values and have to search through all the entries that collided. However, such a collision should be very rare because it occurs only when all 32 bits of the hash value are identical, so most searches require only one comparison.

The binary searching aspect of the digital trie is that it is organized as a 16-way tree that is traversed to find the data. The radix search aspect is that successive 4-bit chunks of the hash value are examined at each level in the tree. The hashing aspect is that it is conceptually a hash table with 2^{32} entries.

Listing III.5. An implementation of digital trie searching.

```
#include        <stdlib.h>
#include        <string.h>
#include        "list.h"
#include        "hash.h"

/*
 * NOTE: This code makes several assumptions about the
 * compiler and machine it is run on. It assumes that
 *
 * 1. The value NULL consists of all "0" bits.
 *
 *      If not, the calloc() call must be changed to
 *      explicitly initialize the pointers allocated.
 *
 * 2. An unsigned and a pointer are the same size.
 *
 *      If not, the use of a union might be incorrect, because
 *      it is assumed that the least significant bit of the
 *      pointer and unsigned members of the union are the
 *      same bit.
 *
 * 3. The least significant bit of a valid pointer
 *    to an object allocated on the heap is always 0.
 *
 *      If not, that bit can't be used as a flag to determine
 *      what type of data the union really holds.
 */

/* number of bits examined at each level of the trie */
```

continues

Listing III.5. continued

```
#define TRIE_BITS       4

/* number of subtries at each level of the trie */
#define TRIE_FANOUT     (1 << TRIE_BITS)

/* mask to get lowest TRIE_BITS bits of the hash */
#define TRIE_MASK       (TRIE_FANOUT - 1)

/*
 * A trie can be either a linked list of elements or
 * a pointer to an array of TRIE_FANOUT tries. The num
 * element is used to test whether the pointer is even
 * or odd.
 */
typedef union trie_u {
        unsigned        num;
        listnode_t    *list;  /* if "num" is even */
        union trie_u    *node;          /* if "num" is odd */
} trie_t;

/*
 * Inserts an element into a trie and returns the resulting
 * new trie. For internal use by trieInsert() only.
 */
static trie_t eleInsert(trie_t t, listnode_t *ele, unsigned h,
                                                int depth)
{
        /*
         * If the trie is an array of tries, insert the
         * element into the proper subtrie.
         */
        if (t.num & 1)
        {
                /*
                 * nxtNode is used to hold the pointer into
                 * the array. The reason for using a trie
                 * as a temporary instead of a pointer is
                 * it's easier to remove the "odd" flag.
                 */
                trie_t  nxtNode = t;

                nxtNode.num &= ~1;
                nxtNode.node += (h >> depth) & TRIE_MASK;
                *nxtNode.node =
                        eleInsert(*nxtNode.node,
                                ele, h, depth + TRIE_BITS);
        }
        /*
         * Since t wasn't an array of tries, it must be a
         * list of elements. If it is empty, just add this
         * element.
         */
        else if (t.list == NULL)
                t.list = ele;
        /*
         * Since the list is not empty, check whether the
         * element belongs on this list or whether you should
```

```
 87:          * make several lists in an array of subtries.
 88:          */
 89:         else if (h == hash(t.list->u.str))
 90:         {
 91:                 ele->next = t.list;
 92:                 t.list = ele;
 93:         }
 94:         else
 95:         {
 96:                 /*
 97:                  * You're making the list into an array or
 98:                  * subtries. Save the current list, replace
 99:                  * this entry with an array of TRIE_FANOUT
100:                  * subtries, and insert both the element and
101:                  * the list in the subtries.
102:                  */
103:                 listnode_t   *lp = t.list;
104:
105:                 /*
106:                  * Calling calloc() rather than malloc()
107:                  * ensures that the elements are initialized
108:                  * to NULL.
109:                  */
110:                 t.node = (trie_t *) calloc(TRIE_FANOUT,
111:                                 sizeof(trie_t));
112:                 t.num |= 1;
113:                 t = eleInsert(t, lp, hash(lp->u.str),
114:                         depth);
115:                 t = eleInsert(t, ele, h, depth);
116:         }
117:         return t;
118: }
119:
120: /*
121:  * Finds an element in a trie and returns the resulting
122:  * string, or NULL. For internal use by search() only.
123:  */
124: static const char * eleSearch(trie_t t, const char * string,
125:                                         unsigned h, int depth)
126: {
127:         /*
128:          * If the trie is an array of subtries, look for the
129:          * element in the proper subtree.
130:          */
131:         if (t.num & 1)
132:         {
133:                 trie_t  nxtNode = t;
134:                 nxtNode.num &= ~1;
135:                 nxtNode.node += (h >> depth) & TRIE_MASK;
136:                 return eleSearch(*nxtNode.node,
137:                         string, h, depth + TRIE_BITS);
138:         }
139:         /*
140:          * Otherwise, the trie is a list. Perform a linear
141:          * search for the desired element.
142:          */
143:         else
```

continues

Listing III.5. continued

```
144:        {
145:                listnode_t   *lp = t.list;
146:
147:                while (lp)
148:                {
149:                        if (strcmp(lp->u.str, string) == 0)
150:                                return lp->u.str;
151:                        lp = lp->next;
152:                }
153:        }
154:        return NULL;
155: }
156:
157: /* Test function to print the structure of a trie */
158: void triePrint(trie_t t, int depth)
159: {
160:        if (t.num & 1)
161:        {
162:                int     i;
163:                trie_t  nxtNode = t;
164:                nxtNode.num &= ~1;
165:                if (depth)
166:                        printf("\n");
167:                for (i = 0; i < TRIE_FANOUT; i++)
168:                {
169:                        if (nxtNode.node[i].num == 0)
170:                                continue;
171:                        printf("%*s[%d]", depth, "", i);
172:                        triePrint(nxtNode.node[i], depth + 8);
173:                }
174:        }
175:        else
176:        {
177:                listnode_t   *lp = t.list;
178:                while (lp)
179:                {
180:                        printf("\t'%s'", lp->u.str);
181:                        lp = lp->next;
182:                }
183:                putchar('\n');
184:        }
185: }
186:
187: static trie_t   t;
188:
189: void insert(const char *s)
190: {
191:        t = eleInsert(t, newNode((void *) s), hash(s), 0);
192: }
193:
194: void print(void)
195: {
196:        triePrint(t, 0);
197: }
```

```
198:
199: const char *search(const char *s)
200: {
201:         return eleSearch(t, s, hash(s), 0);
202: }
```

Cross Reference:

III.4: What is the easiest searching method to use?

III.6: What is hashing?

III.8: How can I search for data in a linked list?

III.6: What is hashing?

Answer:

To hash means to grind up, and that's essentially what hashing is all about. The heart of a hashing algorithm is a hash function that takes your nice, neat data and grinds it into some random-looking integer.

The idea behind hashing is that some data either has no inherent ordering (such as images) or is expensive to compare (such as images). If the data has no inherent ordering, you can't perform comparison searches. If the data is expensive to compare, the number of comparisons used even by a binary search might be too many. So instead of looking at the data themselves, you'll condense (hash) the data to an integer (its hash value) and keep all the data with the same hash value in the same place. This task is carried out by using the hash value as an index into an array.

To search for an item, you simply hash it and look at all the data whose hash values match that of the data you're looking for. This technique greatly lessens the number of items you have to look at. If the parameters are set up with care and enough storage is available for the hash table, the number of comparisons needed to find an item can be made arbitrarily close to one. Listing III.6 shows a simple hashing algorithm. You can combine this example with code at the end of this chapter to produce a working program.

One aspect that affects the efficiency of a hashing implementation is the hash function itself. It should ideally distribute data randomly throughout the entire hash table, to reduce the likelihood of collisions. Collisions occur when two different keys have the same hash value. There are two ways to resolve this problem. In "open addressing," the collision is resolved by the choosing of another position in the hash table for the element inserted later. When the hash table is searched, if the entry is not found at its hashed position in the table, the search continues checking until either the element is found or an empty position in the table is found.

The second method of resolving a hash collision is called "chaining." In this method, a "bucket" or linked list holds all the elements whose keys hash to the same value. When the hash table is searched, the list must be searched linearly.

Listing III.6. A simple example of a hash algorithm.

```
#include        <stdlib.h>
#include        <string.h>
#include        "list.h"
#include        "hash.h"

#define HASH_SIZE       1024

static  listnode_t      *hashTable[HASH_SIZE];

void insert(const char *s)
{
        listnode_t      *ele = newNode((void *) s);
        unsigned int    h = hash(s) % HASH_SIZE;

        ele->next = hashTable[h];
        hashTable[h] = ele;
}

void print(void)
{
        int     h;

        for (h = 0; h < HASH_SIZE; h++)
        {
                listnode_t      *lp = hashTable[h];

                if (lp == NULL)
                        continue;
                printf("[%d]", h);
                while (lp)
                {
                        printf("\t'%s'", lp->u.str);
                        lp = lp->next;
                }
                putchar('\n');
        }
}

const char *search(const char *s)
{
        unsigned int    h = hash(s) % HASH_SIZE;
        listnode_t      *lp = hashTable[h];

        while (lp)
        {
                if (!strcmp(s, lp->u.str))
                        return lp->u.str;
                lp = lp->next;
        }
        return NULL;
}
```

Cross Reference:

III.4: What is the easiest searching method to use?

III.5: What is the quickest searching method to use?

III.8: How can I search for data in a linked list?

III.7: How can I sort a linked list?

Answer:

Both the merge sort and the radix sort shown in FAQ III.2 (see Listings III.2b and III.2c for code) are good sorting algorithms to use for linked lists.

Cross Reference:

III.1: What is the easiest sorting method to use?

III.2: What is the quickest sorting method to use?

III.3: How can I sort things that are too large to bring into memory?

III.8: How can I search for data in a linked list?

Answer:

Unfortunately, the only way to search a linked list is with a linear search, because the only way a linked list's members can be accessed is sequentially. Sometimes it is quicker to take the data from a linked list and store it in a different data structure so that searches can be more efficient.

Cross Reference:

III.4: What is the easiest searching method to use?

III.5: What is the quickest searching method to use?

III.6: What is hashing?

Sample Code

You can combine the following code with the code from each of the listings in this chapter to form a working program you can compile and run. Each example has been compiled and run on the same data set, and the results are compared in the introduction section of this chapter entitled "Performance of Sorting and Searching."

The first listing is a makefile, which can be used with a make utility to compile each program. Because some make utilites don't understand this format, and because not everyone has a make utility, you can use the information in this makefile yourself. Each nonblank line lists the name of an example followed by a colon

and the source files needed to build it. The actual compiler commands will depend on which brand of compiler you have and which options (such as memory model) you want to use. Following the makefile are source files for the main driver programs and the linked list code used by some of the algorithms.

The code in driver1.c (Listing III.9a) sorts all of its command-line arguments, as strings, using whatever sorting algorithm it is built with, and prints the sorted arguments. The code in driver2.c (Listing III.9b) generates a table of the first 10,000 prime numbers, then searches for each of its command-line arguments (which are numbers) in that table.

Because the algorithm in Listing III.6 does not search items in an array, it has its own main procedure. It reads lines of input until it gets to the end-of-file, and then it prints the entire trie data structure. Then it searches for each of its command-line arguments in the trie and prints the results.

Listing III.9. A makefile with rules to build the programs in this chapter.

```
3_1:    3_1.c driver1.c

3_2a:   3_2a.c driver1.c

3_2b:   3_2b.c list.c driver1.c

3_2c:   3_2c.c list.c driver1.c

3_3:    3_3.c 3_1.c

3_4:    3_4.c 3_1.c driver2.c

3_5:    3_5.c 3_1.c driver2.c

3_6:    3_6.c 3_1.c driver2.c

3_7:    3_7.c list.c hash.c driver3.c

3_8:    3_8.c list.c hash.c driver3.c
```

Listing III.9a. The driver1.c driver for all the sorting algorithms except the external sort algorithm.

```
#include        <stdio.h>

extern void sortStrings(const char **);

/* Sorts its arguments and prints them, one per line */
int
main(int argc, const char *argv[])
{
        int     i;

        sortStrings(argv + 1);
        for (i = 1; i < argc; i++)
                puts(argv[i]);
        return 0;
}
```

Listing III.9b. The driver2.c driver for the searching algorithms using `bsearch()`, binary, and linear search algorithms.

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

extern const char       *search(const char *, const char **,
                                                size_t);

static int      size;
static const char       **array;

static void initArray(int limit)
{
        char    buf[1000];

        array = (const char **) calloc(limit, sizeof(char *));
        for (size = 0; size < limit; size++)
        {
                if (gets(buf) == NULL)
                        break;
                array[size] = strdup(buf);
        }
        sortStrings(array, size);
}

int main(int argc, char **argv)
{
        int     i;
        int     limit;

        if (argc < 2)
        {
                fprintf(stderr, "usage: %s size [lookups]\n",
                                                argv[0]);
                exit(1);
        }
        limit = atoi(argv[1]);
        initArray(limit);
        for (i = 2; i < argc; i++)
        {
                const char      *s;

                if (s = search(argv[i], array, limit))
                        printf("%s -> %s\n", argv[i], s);
                else
                        printf("%s not found\n", argv[i]);
        }
        return 0;
}
```

Listing III.9c. The driver3.c driver for the trie and hash search programs.

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

extern void     insert(const char *);
extern void     print(void);
extern const char       *search(const char *);

int
main(int argc, char *argv[])
{
        int     i;
        int     limit;
        char    buf[1000];

        if (argc < 2)
        {
                fprintf(stderr, "usage: %s size [lookups]\n",
                                                argv[0]);
                exit(1);
        }
        limit = atoi(argv[1]);
        for (i = 0; i < limit; i++)
        {
                if (gets(buf) == NULL)
                        break;
                insert(strdup(buf));
        }
        print();
        for (i = 2; i < argc; i++)
        {
                const char      *p = search(argv[i]);
                if (p)
                        printf("%s -> %s\n", argv[i], p);
                else
                        printf("%s not found\n", argv[i]);
        }
        return 0;
}
```

Listing III.9d. The list.h header file, which provides a simple linked list type.

```
/*
 * Generic linked list node structure--can hold either
 * a character string or another list as data.
 */
typedef struct listnode_s {
        struct listnode_s       *next;
        union {
                void    *data;
                struct list_s   *list;
                const char      *str;
        } u;
} listnode_t;
```

```
typedef struct list_s {
        listnode_t      *head;
        listnode_t      *tail;
} list_t;

extern void appendNode(list_t *, listnode_t *);
extern listnode_t *removeHead(list_t *);
extern void concatList(list_t *, list_t *);
extern list_t *copyOf(list_t);
extern listnode_t *newNode(void *);
extern list_t *newList();
```

Listing III.9e. The list.c source file, which provides a simple linked list type.

```
#include        <malloc.h>
#include        "list.h"

/* Appends a listnode_t to a list_t. */
void appendNode(list_t *list, listnode_t *node)
{
        node->next = NULL;
        if (list->head)
        {
                list->tail->next = node;
                list->tail = node;
        }
        else
                list->head = list->tail = node;
}

/* Removes the first node from a list_t and returns it. */
listnode_t *removeHead(list_t *list)
{
        listnode_t      *node = 0;
        if (list->head)
        {
                node = list->head;
                list->head = list->head->next;
                if (list->head == NULL)
                        list->tail = NULL;
                node->next = NULL;
        }
        return node;
}

/* Concatenates two lists into the first list. */
void concatList(list_t *first, list_t *second)
{
        if (first->head)
        {
                if (second->head)
                {
                        first->tail->next = second->head;
                        first->tail = second->tail;
                }
        }
```

continues

Listing III.9e. continued

```
        else
                *first = *second;
        second->head = second->tail = NULL;
}

/* Returns a copy of a list_t from the heap. */
list_t *copyOf(list_t list)
{
        list_t  *new = (list_t *) malloc(sizeof(list_t));
        *new = list;
        return new;
}

/* Allocates a new listnode_t from the heap. */
listnode_t *newNode(void *data)
{
        listnode_t      *new = (listnode_t *)
                                malloc(sizeof(listnode_t));
        new->next = NULL;
        new->u.data = data;
        return new;
}

/* Allocates an empty list_t from the heap. */
list_t *newList()
{
        list_t  *new = (list_t *) malloc(sizeof(list_t));
        new->head = new->tail = NULL;
        return new;
}
```

Listing III.9f. The hash.h header file, which provides a simple character string hash function.

```
unsigned int    hash(const char *);
```

Listing III.9g. The hash.c source file, which provides a simple character string hash function.

```
#include        "hash.h"

/* This is a simple string hash function */
unsigned int hash(const char *string)
{
        unsigned h = 0;

        while (*string)
                h = 17 * h + *string++;
        return h;
}
```