CHAPTER 6

The List Data Model

Like trees, lists are among the most basic of data models used in computer programs. Lists are, in a sense, simple forms of trees, because one can think of a list as a binary tree in which every left child is a leaf. However, lists also present some aspects that are not special cases of what we have learned about trees. For instance, we shall talk about operations on lists, such as pushing and popping, that have no common analog for trees, and we shall talk of character strings, which are special and important kinds of lists requiring their own data structures.

6.1 What This Chapter Is About

We introduce list terminology in Section 6.2. Then in the remainder of the chapter we present the following topics:

- The basic operations on lists (Section 6.3).
- Implementations of abstract lists by data structures, especially the linked-list data structure (Section 6.4) and an array data structure (Section 6.5).
- The stack, a list upon which we insert and delete at only one end (Section 6.6).
- The queue, a list upon which we insert at one end and delete at the other (Section 6.8).
- Character strings and the special data structures we use to represent them (Section 6.10).

Further, we shall study in detail two applications of lists:

- The run-time stack and the way C and many other languages implement recursive functions (Section 6.7).

- The problem of finding longest common subsequences of two strings, and its solution by a "dynamic programming," or table-filling, algorithm (Section 6.9).

6.2 Basic Terminology

List

A *list* is a finite sequence of zero or more elements. If the elements are all of type T, then we say that the type of the list is "list of T." Thus we can have lists of integers, lists of real numbers, lists of structures, lists of lists of integers, and so on. We generally expect the elements of a list to be of some one type. However, since a type can be the union of several types, the restriction to a single "type" can be circumvented.

A list is often written with its elements separated by commas and enclosed in parentheses:

$$(a_1, a_2, \ldots, a_n)$$

where the a_i's are the elements of the list.

Character string

In some situations we shall not show commas or parentheses explicitly. In particular, we shall study *character strings,* which are lists of characters. Character strings are generally written with no comma or other separating marks and with no surrounding parentheses. Elements of character strings will normally be written in typewriter font. Thus `foo` is the list of three characters of which the first is `f` and the second and third are `o`.

Example 6.1. Here are some examples of lists.

1. The list of prime numbers less than 20, in order of size:

   (2, 3, 5, 7, 11, 13, 17, 19)

2. The list of noble gasses, in order of atomic weight:

   (helium, neon, argon, krypton, xenon, radon)

3. The list of the numbers of days in the months of a non-leap year:

   (31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31)

As this example reminds us, the same element can appear more than once on a list. ✦

Example 6.2. A line of text is another example of a list. The individual characters making up the line are the elements of this list, so the list is a character string. This character string usually includes several occurrences of the blank character, and normally the last character in a line of text is the "newline" character.

As another example, a document can be viewed as a list. Here the elements of the list are the lines of text. Thus a document is a list whose elements that are themselves lists, character strings in particular. ✦

✦ **Example 6.3.** A point in n-dimensional space can be represented by a list of n real numbers. For example, the vertices of the unit cube can be represented by the triples shown in Fig. 6.1. The three elements on each list represent the coordinates of a point that is one of the eight corners (or "vertices") of the cube. The first component represents the x-coordinate (horizontal), the second represents the y-coordinate (into the page), and the third represents the z-coordinate (vertical). ✦

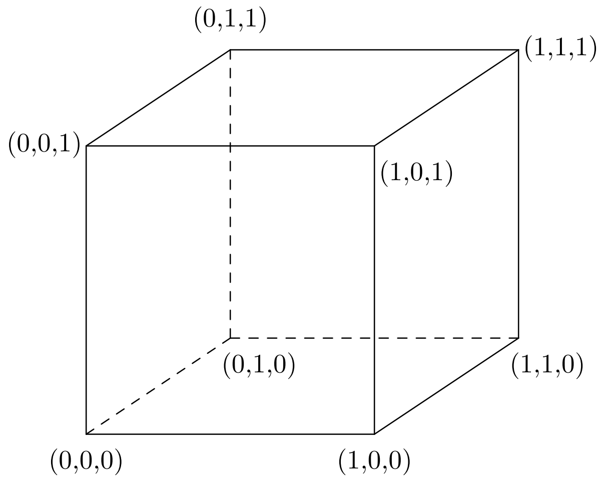


Fig. 6.1. The vertices of the unit cube represented as triples.

The Length of a List

Empty list

The *length* of a list is the number of occurrences of elements on the list. If the number of elements is zero, then the list is said to be *empty.* We use the Greek letter ϵ (epsilon) to represent the empty list. We can also represent the empty list by a pair of parentheses surrounding nothing: (). It is important to remember that length counts positions, not distinct symbols, and so a symbol appearing k times on a list adds k to the length of the list.

✦ **Example 6.4.** The length of list (1) in Example 6.1 is 8, and the length of list (2) is 6. The length of list (3) is 12, since there is one position for each month. The fact that there are only three different numbers on the list is irrelevant as far as the length of the list is concerned. ✦

Parts of a List

Head and tail of a list

If a list is not empty, then it consists of a first element, called the *head* and the remainder of the list, called the *tail.* For instance, the head of list (2) in Example 6.1 is helium, while the tail is the list consisting of the remaining five elements,

(neon, argon, krypton, xenon, radon)

Elements and Lists of Length 1

It is important to remember that the head of a list is an element, while the tail of a list is a list. Moreover, we should not confuse the head of a list — say a — with the list of length 1 containing only the element a, which would normally be written with parentheses as (a). If the element a is of type T, then the list (a) is of type "list of T."

Failure to recognize the difference leads to programming errors when we implement lists by data structures. For example, we may represent lists by linked cells, which are typically structures with an `element` field of some type T, holding an element, and a `next` field holding a pointer to the next cell. Then element a is of type T, while the list (a) is a cell with `element` field holding a and `next` field holding `NULL`.

Sublist

If $L = (a_1, a_2, \ldots, a_n)$ is a list, then for any i and j such that $1 \leq i \leq j \leq n$, $(a_i, a_{i+1}, \ldots, a_j)$ is said to be a *sublist* of L. That is, a sublist is formed by starting at some position i, and taking all the elements up to some position j. We also say that ϵ, the empty list, is a sublist of any list.

Subsequence

A *subsequence* of the list $L = (a_1, a_2, \ldots, a_n)$ is a list formed by striking out zero or more elements of L. The remaining elements, which form the subsequence, must appear in the same order in which they appear in L, but the elements of the subsequence need not be consecutive in L. Note that ϵ and the list L itself are always subsequences, as well as sublists, of L.

✦ **Example 6.5.** Let L be the character string `abc`. The sublists of L are

ϵ, `a`, `b`, `c`, `ab`, `bc`, `abc`

These are all subsequences of L as well, and in addition, `ac` is a subsequence, but not a sublist.

For another example, let L be the character string `abab`. Then the sublists are

ϵ, `a`, `b`, `ab`, `ba`, `aba`, `bab`, `abab`

These are also subsequences of L, and in addition, L has the subsequences `aa`, `bb`, `aab`, and `abb`. Notice that a character string like `bba` is not a subsequence of L. Even though L has two `b`'s and an `a`, they do not appear in such an order in L that we can form `bba` by striking out elements of L. That is, there is no `a` after the second `b` in L. ✦

Prefix and suffix

A *prefix* of a list is any sublist that starts at the beginning of the list. A *suffix* is a sublist that terminates at the end of the list. As special cases, we regard ϵ as both a prefix and a suffix of any list.

✦ **Example 6.6.** The prefixes of the list `abc` are ϵ, `a`, `ab`, and `abc`. Its suffixes are ϵ, `c`, `bc`, and `abc`. ✦

Car and Cdr

In the programming language Lisp, the head is called the *car* and the tail is called the *cdr* (pronounced "cudder"). The terms "*car*" and "*cdr*" arose from the names given to two fields of a machine instruction on an IBM 709, the computer on which Lisp was first implemented. Car stands for "contents of the address register," and cdr stands for "contents of the decrement register." In a sense, memory words were seen as cells with `element` and `next` fields, corresponding to the car and cdr, respectively.

The Position of an Element on a List

Each element on a list is associated with a *position.* If $(a_1, a_2, \ldots, a_n)$ is a list and $n \geq 1$, then a_1 is said to be *first* element, a_2 the second, and so on, with a_n the *last* element. We also say that element a_i occurs at *position* i. In addition, a_i is said to *follow* a_{i-1} and to *precede* a_{i+1}. A position holding element a is said to be an *occurrence* of a.

Occurrence of an element

The number of positions on a list equals the length of the list. It is possible for the same element to appear at two or more positions. Thus it is important not to confuse a position with the element at that position. For instance, list (3) in Example 6.1 has twelve positions, seven of which hold 31 — namely, positions 1, 3, 5, 7, 8, 10, and 12.

EXERCISES

6.2.1: Answer the following questions about the list $(2, 7, 1, 8, 2)$.

a) What is the length?
b) What are all the prefixes?
c) What are all the suffixes?
d) What are all the sublists?
e) How many subsequences are there?
f) What is the head?
g) What is the tail?
h) How many positions are there?

6.2.2: Repeat Exercise 6.2.1 for the character string `banana`.

6.2.3**: In a list of length $n \geq 0$, what are the largest and smallest possible numbers of (a) prefixes (b) sublists (c) subsequences?

6.2.4: If the tail of the tail of the list L is the empty list, what is the length of L?

6.2.5*: Bea Fuddled wrote a list whose elements are themselves lists of integers, but omitted the parentheses: 1,2,3. There are many lists of lists that could have been represented, such as $\big((1), (2, 3)\big)$. What are all the possible lists that do not have the empty list as an element?

6.3 Operations on Lists

A great variety of operations can be performed on lists. In Chapter 2, when we discussed merge sort, the basic problem was to sort a list, but we also needed to split a list into two, and to merge two sorted lists. Formally, the operation of *sorting* a list $(a_1, a_2, \dots, a_n)$ amounts to replacing the list with a list consisting of a permutation of its elements, $(b_1, b_2, \dots, b_n)$, such that $b_1 \le b_2 \le \cdots \le b_n$. Here, as before, $\le$ represents an ordering of the elements, such as "less than or equal to" on integers or reals, or lexicographic order on strings. The operation of *merging* two sorted lists consists of constructing from them a sorted list containing the same elements as the two given lists. Multiplicity must be preserved; that is, if there are k occurrences of element a among the two given lists, then the resulting list has k occurrences of a. Review Section 2.8 for examples of these two operations on lists.

Sorting and merging

Insertion, Deletion, and Lookup

Recall from Section 5.7 that a "dictionary" is a set of elements on which we perform the operations *insert*, *delete*, and *lookup*. There is an important difference between sets and lists. An element can never appear more than once in a set, although, as we have seen, an element can appear more than once on a list; this and other issues regarding sets are discussed in Chapter 7. However, lists can implement sets, in the sense that the elements in a set $\{a_1, a_2, \dots, a_n\}$ can be placed in a list in any order, for example, the order $(a_1, a_2, \dots, a_n)$, or the order $(a_n, a_{n-1}, \dots, a_1)$. Thus it should be no surprise that there are operations on lists analogous to the dictionary operations on sets.

1. We can *insert* an element x onto a list L. In principle, x could appear anywhere on the list, and it does not matter if x already appears in L one or more times. We insert by adding one more occurrence of x. As a special case, if we make x the head of the new list (and therefore make L the tail), we say that we *push* x onto the list L. If $L = (a_1, \dots, a_n)$, the resulting list is $(x, a_1, \dots, a_n)$.

2. We can *delete* an element x from a list L. Here, we delete an occurrence of x from L. If there is more than one occurrence, we could specify which occurrence to delete; for example, we could always delete the first occurrence. If we want to delete all occurrences of x, we repeat the deletion until no more x's remain. If x is not present on list L, the deletion has no effect. As a special case, if we delete the head element of the list, so that the list $(x, a_1, \dots, a_n)$ becomes $(a_1, \dots, a_n)$, we are said to *pop* the list.

3. We can *lookup* an element x on a list L. This operation returns `TRUE` or `FALSE`, depending on whether x is or is not an element on the list.

✦ **Example 6.7.** Let L be the list $(1, 2, 3, 2)$. The result of $insert(1, L)$ could be the list $(1, 1, 2, 3, 2)$, if we chose to push 1, that is, to insert 1 at the beginning. We could also insert the new 1 at the end, yielding $(1, 2, 3, 2, 1)$. Alternatively, the new 1 could be placed in any of three positions interior to the list L.

The result of $delete(2, L)$ is the list $(1, 3, 2)$ if we delete the first occurrence of 2. If we ask $lookup(x, L)$, the answer is `TRUE` if x is 1, 2, or 3, but `FALSE` if x is any other value. ✦

Concatenation

We *concatenate* two lists L and M by forming the list that begins with the elements of L and then continues with the elements of M. That is, if $L = (a_1, a_2, \ldots, a_n)$ and $M = (b_1, b_2, \ldots, b_k)$, then LM, the concatenation of L and M, is the list

$$(a_1, a_2, \ldots, a_n, b_1, b_2, \ldots, b_k)$$

Identity for concatenation

Note that the empty list is the identity for concatenation. That is, for any list L, we have $\epsilon L = L\epsilon = L$.

✦ **Example 6.8.** If L is the list $(1, 2, 3)$ and M is the list $(3, 1)$, then LM is the list $(1, 2, 3, 3, 1)$. If L is the character string `dog` and M is the character string `house`, then LM is the character string `doghouse`. ✦

Other List Operations

Another family of operations on lists refers to particular positions of the list. For example,

First and last of lists

a) The operation $first(L)$ returns the first element (head) of list L, and $last(L)$ returns the last element of L. Both cause an error if L is an empty list.

Retrieve

b) The operation $retrieve(i, L)$ returns the element at the ith position of list L. It is an error if L has length less than i.

There are additional operations that involve the length of a list. Some common operations are:

c) $length(L)$, which returns the length of list L.

isEmpty

d) $isEmpty(L)$, which returns `TRUE` if L is an empty list and returns `FALSE` if not. Similarly, $isNotEmpty(L)$ would return the opposite.

EXERCISES

6.3.1: Let L be the list $(3, 1, 4, 1, 5, 9)$.

a) What is the value of $delete(5, L)$?
b) What is the value of $delete(1, L)$?
c) What is the result of popping L?
d) What is the result of pushing 2 onto list L?
e) What is returned if we perform *lookup* with the element 6 and list L?
f) If M is the list $(6, 7, 8)$, what is the value of LM (the concatenation of L and M)? What is ML?
g) What is $first(L)$? What is $last(L)$?
h) What is the result of $retrieve(3, L)$?
i) What is the value of $length(L)$?
j) What is the value of $isEmpty(L)$?

6.3.2**: If L and M are lists, under what conditions is $LM = ML$?

6.3.3**: Let x be an element and L a list. Under what conditions are the following equations true?

a) $delete\big(x,\ insert(x, L)\big) = L$
b) $insert\big(x,\ delete(x, L)\big) = L$
c) $first(L) = retrieve(1, L)$
d) $last(L) = retrieve\big(length(L),\ L\big)$

6.4 The Linked-List Data Structure

The easiest way to implement a list is to use a linked list of cells. Each cell consists of two fields, one containing an element of the list, the other a pointer to the next cell on the linked list. In this chapter we shall make the simplifying assumption that elements are integers. Not only may we use the specific type `int` for the type of elements, but we can compare elements by the standard comparison operators `==`, `<`, and so on. The exercises invite the reader to develop variants of our functions that work for arbitrary types, where comparisons are made by user-defined functions such as *eq* to test equality, $lt(x, y)$ to test if x precedes y in some ordering, and so on.

In what follows, we shall use our macro from Section 1.6:

```
DefCell(int, CELL, LIST);
```

which expands into our standard structure for cells and lists:

```
typedef struct CELL *LIST;
struct CELL {
    int element;
    LIST next;
};
```

Note that `LIST` is the type of a pointer to a cell. In effect, the `next` field of each cell points both to the next cell and to the entire remainder of the list.

Figure 6.2 shows a linked list that represents the abstract list

$$L = (a_1, a_2, \ldots, a_n)$$

There is one cell for each element; the element a_i appears in the `element` field of the ith cell. The pointer in the ith cell points to the $(i+1)$st cell, for $i = 1, 2, \ldots, n-1$, and the pointer in the last cell is `NULL`, indicating the end of the list. Outside the list is a pointer, named `L`, that points to the first cell of the list; `L` is of type `LIST`. If the list L were empty, then the value of `L` would be `NULL`.

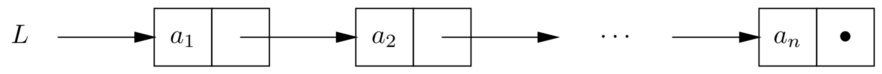


Fig. 6.2. A linked list representing the list $L = (a_1, a_2, \ldots, a_n)$.

Implementation of Dictionary Operations by Linked Lists

Let us consider how we can implement the dictionary operations if we represent the dictionary by a linked list. The following operations on dictionaries were defined in Section 5.7.

Lists and Linked Lists

Remember that a list is an abstract, or mathematical model. The linked list is a simple data structure, which was mentioned in Chapter 1. A linked list is one way to implement the list data model, although, as we shall see, it is not the only way. At any rate, this is a good time to remember once more the distinction between models and the data structures that implement them.

1. *insert*(x, D), to insert element x into dictionary D,
2. *delete*(x, D), to delete element x from dictionary D, and
3. *lookup*(x, D), to determine whether element x is in dictionary D,

We shall see that the linked list is a simpler data structure for implementing dictionaries than the binary search tree that we discussed in the previous chapter. However, the running time of the dictionary operations when using the linked-list representation is not as good as when using the binary search tree. In Chapter 7 we shall see an even better implementation for dictionaries — the hash table — which makes use, as subroutines, of the dictionary operations on lists.

We shall assume that our dictionary contains integers, and cells are defined as they were at the beginning of this section. Then the type of a dictionary is `LIST`, also as defined at the beginning of this section. The dictionary containing the set of elements $\{a_1, a_2, \ldots, a_n\}$ could be represented by the linked list of Fig. 6.2. There are many other lists that could represent the same set, since order of elements is not important in sets.

Lookup

To perform *lookup*(x, D), we examine each cell of the list representing D to see whether it holds the desired element x. If so, we return `TRUE`. If we reach the end of the list without finding x, we return `FALSE`. As before, the defined constants `TRUE` and `FALSE` stand for the constants 1 and 0, and `BOOLEAN` for the defined type `int`. A recursive function `lookup(x,D)` is shown in Fig. 6.3.

```
BOOLEAN lookup(int x, LIST L)
{
    if (L == NULL)
        return FALSE;
    else if (x == L->element)
        return TRUE;
    else
        return lookup(x, L->next);
}
```

Fig. 6.3. Lookup on a linked list.

If the list has length n, we claim that the function of Fig. 6.3 takes $O(n)$ time. Except for the recursive call at the end, `lookup` takes $O(1)$ time. When the call is made, the length of the remaining list is 1 less than the length of the list L. Thus it should surprise no one that `lookup` on a list of length n takes $O(n)$ time. More formally, the following recurrence relation gives the running time $T(n)$ of `lookup` when the list L pointed to by the second argument has length n.

BASIS. $T(0)$ is $O(1)$, because there is no recursive call when L is `NULL`.

INDUCTION. $T(n) = T(n-1) + O(1)$.

The solution to this recurrence is $T(n) = O(n)$, as we saw several times in Chapter 3. Since a dictionary of n elements is represented by a list of length n, *lookup* takes $O(n)$ time on a dictionary of size n.

Unfortunately, the average time for a successful lookup is also proportional to n. For if we are looking for an element x known to be in D, the expected value of the position of x in the list is $(n+1)/2$. That is, x could be anywhere from the first to the nth element, with equal probability. Thus, the expected number of recursive calls to `lookup` is $(n+1)/2$. Since each takes $O(1)$ time, the average successful lookup takes $O(n)$ time. Of course, if the lookup is unsuccessful, we make all n calls before reaching the end of the list and returning `FALSE`.

Deletion

A function to delete an element x from a linked list is shown in Fig. 6.4. The second parameter `pL` is a pointer to the list L (rather than the list L itself). We use the "call by reference" style here because we want `delete` to remove the cell containing x from the list. As we move down the list, `pL` holds a pointer to a pointer to the "current" cell. If we find x in the current cell C, at line (2), then we change the pointer to cell C at line (3), so that it points to the cell following C on the list. If C happens to be last on the list, the former pointer to C becomes `NULL`. If x is not the current element, then at line (4) we recursively delete x from the tail of the list.

Note that the test at line (1) causes the function to return with no action if the list is empty. That is because x is not present on an empty list, and we need not do anything to remove x from the dictionary. If D is a linked list representing a dictionary, then a call to `delete(x, &D)` initiates the deletion of x from the dictionary D.

```
        void delete(int x, LIST *pL)
        {
(1)         if ((*pL) != NULL)
(2)             if (x == (*pL)->element)
(3)                 (*pL) = (*pL)->next;
                else
(4)                 delete(x, &((*pL)->next));
        }
```

Fig. 6.4. Deleting an element.

If the element x is not on the list for the dictionary D, then we run down to the end of the list, taking $O(1)$ time for each element. The analysis is similar to that for the `lookup` function of Fig. 6.3, and we leave the details for the reader. Thus the time to delete an element not in D is $O(n)$ if D has n elements. If x is in the dictionary D, then, on the average, we shall encounter x halfway down the list. Therefore we search $(n+1)/2$ cells on the average, and the running time of a successful deletion is also $O(n)$.

Insertion

A function to insert an element x into a linked list is shown in Fig. 6.5. To insert x, we need to check that x is not already on the list (if it is, we do nothing). If x is not already present, we must add it to the list. It does not matter where in the list we add x, but the function in Fig. 6.5 adds x to the end of the list. When at line (1) we detect the `NULL` at the end, we are therefore sure that x is not already on the list. Then, lines (2) through (4) append x to the end of the list.

If the list is not `NULL`, line (5) checks for x at the current cell. If x is not there, line (6) makes a recursive call on the tail. If x is found at line (5), then function `insert` terminates with no recursive call and with no change to the list L. A call to `insert(x, &D)` initiates the insertion of x into dictionary D.

```
          void insert(int x, LIST *pL)
          {
(1)           if ((*pL) == NULL) {
(2)               (*pL) = (LIST) malloc(sizeof(struct CELL));
(3)               (*pL)->element = x;
(4)               (*pL)->next = NULL;
              }
(5)           else if (x != (*pL)->element)
(6)               insert(x, &((*pL)->next));
          }
```

Fig. 6.5. Inserting an element.

As in the case of lookup and deletion, if we do not find x on the list, we travel to the end, taking $O(n)$ time. If we do find x, then on the average we travel halfway[1] down the list, and we still take $O(n)$ time on the average.

A Variant Approach with Duplicates

We can make insertion run faster if we do not check for the presence of x on the list before inserting it. However, as a consequence, there may be several copies of an element on the list representing a dictionary.

To execute the dictionary operation $insert(x, D)$ we simply create a new cell, put x in it, and push that cell onto the front of the list for D. This operation takes $O(1)$ time.

The lookup operation is exactly the same as in Fig. 6.3. The only nuance is that we may have to search a longer list, because the length of the list representing dictionary D may be greater than the number of members of D.

[1] In the following analyses, we shall use terms like "halfway" or "$n/2$" when we mean the middle of a list of length n. Strictly speaking, $(n+1)/2$ is more accurate.

Abstraction Versus Implementation Again

It may be surprising to see us using duplicates in lists that represent dictionaries, since the abstract data type DICTIONARY is defined as a set, and sets do not have duplicates. However, it is not the dictionary that has duplicates. Rather the data structure implementing the dictionary is allowed to have duplicates. But even when x appears several times on a linked list, it is only present once in the dictionary that the linked list represents.

Deletion is slightly different. We cannot stop our search for x when we encounter a cell with element x, because there could be other copies of x. Thus we must delete x from the tail of a list L, even when the head of L contains x. As a result, not only do we have longer lists to contend with, but to achieve a successful deletion we must search every cell rather than an average of half the list, as we could for the case in which no duplicates were allowed on the list. The details of these versions of the dictionary operations are left as an exercise.

In summary, by allowing duplicates, we make insertion faster, $O(1)$ instead of $O(n)$. However, successful deletions require search of the entire list, rather than an average of half the list, and for both lookup and deletion, we must contend with lists that are longer than when duplicates are not allowed, although how much longer depends on how often we insert an element that is already present in the dictionary.

Which method to choose is a bit subtle. Clearly, if insertions predominate, we should allow duplicates. In the extreme case, where we do only insertions but never lookup or deletion, we get performance of $O(1)$ per operation, instead of $O(n)$.[2] If we can be sure, for some reason, that we shall never insert an element already in the dictionary, then we can use both the fast insertion and the fast deletion, where we stop when we find one occurrence of the element to be deleted. On the other hand, if we may insert duplicate elements, and lookups or deletions predominate, then we are best off checking for the presence of x before inserting it, as in the *insert* function of Fig. 6.5.

Sorted Lists to Represent Dictionaries

Another alternative is to keep elements sorted in increasing order on the list representing a dictionary D. Then, if we wish to lookup element x, we have only to go as far as the position in which x would appear; on the average, that is halfway down the list. If we meet an element greater than x, then there is no hope of finding x later in the list. We thus avoid going all the way down the list on unsuccessful searches. That saves us about a factor of 2, although the exact factor is somewhat clouded because we have to ask whether x follows in sorted order each of the elements we meet on the list, which is an additional step at each cell. However, the same factor in savings is gained on unsuccessful searches during insertion and deletion.

A lookup function for sorted lists is shown in Fig. 6.6. We leave to the reader the exercise of modifying the functions of Figs. 6.4 and 6.5 to work on sorted lists.

[2] But why bother inserting into a dictionary if we never look to see what is there?

```
BOOLEAN lookup(int x, LIST L)
{
    if (L == NULL)
        return FALSE;
    else if (x > L->element)
        return lookup(x, L->next);
    else if (x == L->element)
        return TRUE;
    else /* here x < L->element, and so x could not be
            on the sorted list L */
        return FALSE;
}
```

Fig. 6.6. Lookup on a sorted list.

Comparison of Methods

The table in Fig. 6.7 indicates the number of cells we must search for each of the three dictionary operations, for each of the three list-based representations of dictionaries we have discussed. We take n to be the number of elements in the dictionary, which is also the length of the list if no duplicates are allowed. We use m for the length of the list when duplicates are allowed. We know that $m \geq n$, but we do not know how much greater m is than n. Where we use $n/2 \rightarrow n$ we mean that the number of cells is an average of $n/2$ when the search is successful, and n when unsuccessful. The entry $n/2 \rightarrow m$ indicates that on a successful lookup we shall see $n/2$ elements of the dictionary, on the average, before seeing the one we want,[3] but on an unsuccessful search, we must go all the way to the end of a list of length m.

	INSERT	DELETE	LOOKUP
No duplicates	$n/2 \rightarrow n$	$n/2 \rightarrow n$	$n/2 \rightarrow n$
Duplicates	0	m	$n/2 \rightarrow m$
Sorted	$n/2$	$n/2$	$n/2$

Fig. 6.7. Number of cells searched by three methods of representing dictionaries by linked lists.

Notice that all of these running times, except for insertion with duplicates, are worse than the average running times for dictionary operations when the data structure is a binary search tree. As we saw in Section 5.8, dictionary operations take only $O(\log n)$ time on the average when a binary search tree is used.

[3] In fact, since there may be duplicates, we may have to examine somewhat more than $n/2$ cells before we can expect to see $n/2$ different elements.

Judicious Ordering of Tests

Notice the order in which the three tests of Fig. 6.6 are made. We have no choice but to test that L is not `NULL` first, since if L is `NULL` the other two tests will cause an error. Let y be the value of `L->element`. Then in all but the last cell we visit, we shall have $x < y$. The reason is that if we have $x = y$, we terminate the lookup successfully, and if we have $x > y$, we terminate with failure to find x. Thus we make the test $x < y$ first, and only if that fails do we need to separate the other two cases. That ordering of tests follows a general principle: we want to test for the most common cases first, and thus save in the total number of tests we perform, on the average.

If we visit k cells, then we test k times whether L is `NULL` and we test k times whether x is less than y. Once, we shall test whether $x = y$, making a total of $2k + 1$ tests. That is only one more test than we make in the `lookup` function of Fig. 6.3 — which uses unsorted lists — in the case that the element x is found. If the element is not found, we shall expect to use many fewer tests in Fig. 6.6 than in Fig. 6.3, because we can, on the average, stop after examining only half the cells with Fig. 6.6. Thus although the big-oh running times of the dictionary operations using either sorted or unsorted lists is $O(n)$, there is usually a slight advantage in the constant factor if we use sorted lists.

Doubly Linked Lists

In a linked list it is not easy to move from a cell toward the beginning of the list. The doubly linked list is a data structure that facilitates movement both forward and backward in a list. The cells of a doubly linked list of integers contain three fields:

```
typdef struct CELL *LIST;
struct CELL {
    LIST previous;
    int element;
    LIST next;
};
```

The additional field contains a pointer to the previous cell on the list. Figure 6.8 shows a doubly linked list data structure that represents the list $L = (a_1, a_2, \ldots, a_n)$.

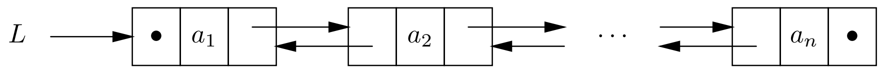


Fig. 6.8. A doubly linked list representing the list $L = (a_1, a_2, \ldots, a_n)$.

Dictionary operations on a doubly linked list structure are essentially the same as those on a singly linked list. To see the advantage of doubly linked lists, consider the operation of deleting an element a_i, given only a pointer to the cell containing that element. With a singly linked list, we would have to find the previous cell by searching the list from the beginning. With a doubly linked list, we can do the task in $O(1)$ time by a sequence of pointer manipulations, as shown in Fig. 6.9.

```
          void delete(LIST p, LIST *pL)
          {
              /* p is a pointer to the cell to be deleted,
                and pL points to the list */
   (1)        if (p->next != NULL)
   (2)            p->next->previous = p->previous;
   (3)        if (p->previous == NULL) /* p points to first cell */
   (4)            (*pL) = p->next;
              else
   (5)            p->previous->next = p->next;
          }
```

Fig. 6.9. Deleting a cell from a doubly linked list.

The function `delete(p,pL)` shown in Fig. 6.9 takes as arguments a pointer p to the cell to be deleted, and pL, which is a pointer to the list L itself. That is, pL is the address of a pointer to the first cell on the list. In line (1) of Fig. 6.9 we check that p does not point to the last cell. If it does not, then at line (2) we make the backward pointer of the following cell point to the cell before p (or we make it equal to `NULL` if p happens to point to the first cell).

Line (3) tests whether p is the first cell. If so, then at line (4) we make L point to the second cell. Note that in this case, line (2) has made the `previous` field of the second cell `NULL`. If p does not point to the first cell, then at line (5) we make the forward pointer of the previous cell point to the cell following p. That way, the cell pointed to by p has effectively been spliced out of the list; the previous and next cells point to each other.

EXERCISES

6.4.1: Set up the recurrence relations for the running times of (a) `delete` in Fig. 6.4 (b) `insert` in Fig. 6.5. What are their solutions?

6.4.2: Write C functions for dictionary operations *insert*, *lookup* and *delete* using linked lists with duplicates.

6.4.3: Write C functions for *insert* and *delete*, using sorted lists as in Fig. 6.6.

6.4.4: Write a C function that inserts an element x into a new cell that follows the cell pointed to by p on a doubly linked list. Figure 6.9 is a similar function for deletion, but for insertion, we don't need to know the list header L.

6.4.5: If we use the doubly linked data structure for lists, an option is to represent a list not by a pointer to a cell, but by a cell with the element field unused. Note that this "header" cell is not itself a part of the list. The `next` field of the header points to the first true cell of the list, and the `previous` field of the first cell points to the header cell. We can then delete the cell (not the header) pointed to by pointer p without knowing the header L, as we needed to know in Fig. 6.9. Write a C function to delete from a doubly linked list using the format described here.

6.4.6: Write recursive functions for (a) $retrieve(i, L)$ (b) $length(L)$ (c) $last(L)$ using the linked-list data structure.

6.4.7: Extend each of the following functions to cells with an arbitrary type `ETYPE` for elements, using functions $eq(x, y)$ to test if x and y are equal and $lt(x, y)$ to tell if x precedes y in an ordering of the elements of `ETYPE`.

a) *lookup* as in Fig. 6.3.
b) *delete* as in Fig. 6.4.
c) *insert* as in Fig. 6.5.
d) *insert*, *delete*, and *lookup* using lists with duplicates.
e) *insert*, *delete*, and *lookup* using sorted lists.

6.5 Array-Based Implementation of Lists

Another common way to implement a list is to create a structure consisting of

1. An array to hold the elements and
2. A variable `length` to keep track of the count of the number of elements currently in the list.

Figure 6.10 shows how we might represent the list $(a_0, a_1, \ldots, a_{n-1})$ using an array `A[0..MAX-1]`. Elements a_0, $a_1, \ldots, a_{n-1}$ are stored in `A[0..n-1]`, and $length = n$.

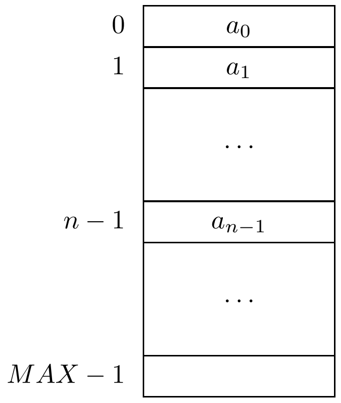


Fig. 6.10. The array `A` holding the list $(a_0, a_1, \ldots, a_{n-1})$.

As in the previous section, we assume that list elements are integers and invite the reader to generalize the functions to arbitrary types. The structure declaration for this array-based implementation of lists is:

```
typedef struct {
    int A[MAX];
    int length;
} LIST;
```

Here, `LIST` is a structure of two fields; the first is an array `A` that stores the elements, the second an integer `length` that contains the number of elements currently on the list. The quantity `MAX` is a user-defined constant that bounds the number of elements that will ever be stored on the list.

The array-based representation of lists is in many ways more convenient than the linked-list representation. It does suffer, however, from the limitation that lists cannot grow longer than the array, which can cause an insertion to fail. In the linked-list representation, we can grow lists as long as we have available computer memory.

We can perform the dictionary operations on array-based lists in roughly the same time as on lists in the linked-list representation. To insert x, we look for x, and if we do not find it, we check whether $length < MAX$. If not, there is an error condition, as we cannot fit the new element into the array. Otherwise, we store x in `A[length]` and then increase *length* by 1. To delete x, we again lookup x, and if found, we shift all following elements of `A` down by one position; then, we decrement *length* by 1. The details of functions to implement *insert* and *delete* are left as exercises. We shall concentrate on the details of *lookup* below.

Lookup with Linear Search

Figure 6.11 is a function that implements the operation *lookup*. Because the array `A` may be large, we choose to pass a pointer `pL` to the structure of type `LIST` as a formal parameter to *lookup*. Within the function, the two fields of the structure can then be referred to as `pL->A[i]` and `pL->length`.

Starting with $i = 0$, the for-loop of lines (1) to (3) examines each location of the array in turn, until it either reaches the last occupied location or finds x. If it finds x, it returns `TRUE`. If it has examined each element in the list without finding x, it returns `FALSE` at line (4). This method of lookup is called *linear* or *sequential* search.

```
        BOOLEAN lookup(int x, LIST *pL)
        {
            int i;

(1)         for (i = 0; i < pL->length; i++)
(2)             if (x == pL->A[i])
(3)                 return TRUE;
(4)         return FALSE;
        }
```

Fig. 6.11. Function that does lookup by linear search.

It is easy to see that, on the average, we search half the array `A[0..length-1]` before finding x if it is present. Thus letting n be the value of `length`, we take $O(n)$ time to perform a lookup. If x is not present, we search the whole array `A[0..length-1]`, again requiring $O(n)$ time. This performance is the same as for a linked-list representation of a list.

The Importance of Constant Factors in Practice

Throughout Chapter 3, we emphasized the importance of big-oh measures of running time, and we may have given the impression that big-oh is all that matters or that any $O(n)$ algorithm is as good as any other $O(n)$ algorithm for the same job. Yet here, in our discussion of sentinels, and in other sections, we have examined rather carefully the constant factor hidden by the $O(n)$. The reason is simple. While it is true that big-oh measures of running time dominate constant factors, it is also true that everybody studying the subject learns that fairly quickly. We learn, for example, to use an $O(n \log n)$ running time sort whenever n is large enough to matter. A competitive edge in the performance of software frequently comes from improving the constant factor in an algorithm that already has the right "big-oh" running time, and this edge frequently translates into the success or failure of a commercial software product.

Lookup with Sentinels

We can simplify the code in the for-loop of Fig 6.11 and speed up the program by temporarily inserting x at the end of the list. This x at the end of the list is called a *sentinel*. The technique was first mentioned in a box on "More Defensive Programming" in Section 3.6, and it has an important application here. Assuming that there always is an extra slot at the end of the list, we can use the program in Fig. 6.12 to search for x. The running time of the program is still $O(n)$, but the constant of proportionality is smaller because the number of machine instructions required by the body and test of the loop will typically be smaller for Fig. 6.12 than for Fig. 6.11.

```
        BOOLEAN lookup(int x, LIST *pL)
        {
            int i;

(1)         pL->A[pL->length] = x;
(2)         i = 0;
(3)         while (x != pL->A[i])
(4)             i++;
(5)         return (i < pL->length);
        }
```

Fig. 6.12. Function that does lookup with a sentinel.

The sentinel is placed just beyond the list by line (1). Note that since *length* does not change, this x is not really part of the list. The loop of lines (3) and (4) increases i until we find x. Note that we are guaranteed to find x even if the list is empty, because of the sentinel. After finding x, we test at line (5) whether we have found a real occurrence of x (that is, $i < length$), or whether we have found the sentinel (that is, $i = length$). Note that if we are using a sentinel, it is essential that *length* be kept strictly less than *MAX*, or else there will be no place to put the sentinel.

Lookup on Sorted Lists with Binary Search

Suppose L is a list in which the elements $a_0, a_1, \ldots, a_{n-1}$ have been sorted in non-decreasing order. If the sorted list is stored in an array `A[0..n-1]`, we can speed lookups considerably by using a technique known as *binary search*. We must first find the index m of the middle element; that is, $m = \lfloor (n-1)/2 \rfloor$.[4] Then we compare element x with $A[m]$. If they are equal, we have found x. If $x < A[m]$, we recursively repeat the search on the sublist `A[0..m-1]`. If $x > A[m]$, we recursively repeat the search on the sublist `A[m+1..n-1]`. If at any time we try to search an empty list, we report failure. Figure 6.13 illustrates the division process.

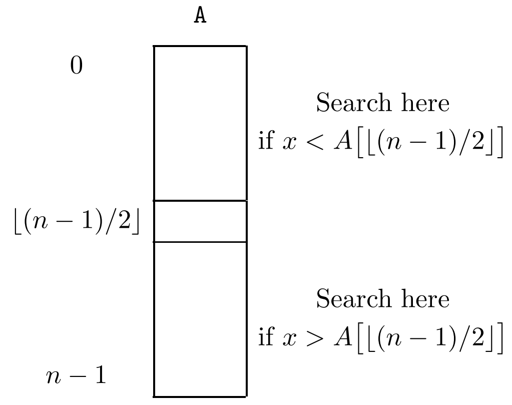

A
0
Search here
if $x < A\big[\lfloor (n-1)/2 \rfloor\big]$
$\lfloor (n-1)/2 \rfloor$
Search here
if $x > A\big[\lfloor (n-1)/2 \rfloor\big]$
$n-1$

Fig. 6.13. Binary search divides a range in two.

The code for a function *binsearch* to locate x in a sorted array A is shown in Fig. 6.14. The function uses the variables `low` and `high` for the lower and upper bounds of the range in which x might lie. If the lower range exceeds the upper, we have failed to find x; the function terminates and returns `FALSE`.

Otherwise, *binsearch* computes the midpoint of the range, by

$$mid = \lfloor (low + high)/2 \rfloor$$

Then the function examines $A[mid]$, the element at the middle of the range, to determine whether x is there. If not, it continues the search in the lower or upper half of the range, depending on whether x is less than or greater than $A[mid]$. This idea generalizes the division suggested in Fig. 6.13, where *low* was 0 and *high* was $n-1$.

The function `binsearch` can be proved correct using the inductive assertion that if x is in the array, then it must lie within the range `A[low..high]`. The proof is by induction on the difference $high - low$ and is left as an exercise.

At each iteration, *binsearch* either

1. Finds the element x when it reaches line (8) or

[4] The notation $\lfloor a \rfloor$, the *floor* of a, is the integer part of a. Thus $\lfloor 6.5 \rfloor = 6$ and $\lfloor 6 \rfloor = 6$. Also, $\lceil a \rceil$, the *ceiling* of a, is the smallest integer greater than or equal to a. For instance, $\lceil 6.5 \rceil = 7$ and $\lceil 6 \rceil = 6$.

```
        BOOLEAN binsearch(int x, int A[], int low, int high)
        {
            int mid;

(1)         if (low > high)
(2)             return FALSE;
            else {
(3)             mid = (low + high)/2;
(4)             if (x < A[mid])
(5)                 return binsearch(x, A, low, mid-1);
(6)             else if (x > A[mid])
(7)                 return binsearch(x, A, mid+1, high);
                else /* x == A[mid] */
(8)                 return TRUE;
            }
        }
```

Fig. 6.14. Function that does lookup using binary search.

2. Calls itself recursively at line (5) or line (7) on a sublist that is at most half as long as the array `A[low..high]` that it was given to search.

Starting with an array of length n, we cannot divide the length of the array to be searched in half more than $\log_2 n$ times before it has length 1, whereupon we either find x at `A[mid]`, or we fail to find x at all after a call on the empty list.

To look for x in an array `A` with n elements, we call `binsearch(x,A,0,n-1)`. We see that `binsearch` calls itself $O(\log n)$ times at most. At each call, we spend $O(1)$ time, plus the time of the recursive call. The running time of binary search is therefore $O(\log n)$. That compares favorably with the linear search, which takes $O(n)$ time on the average, as we have seen.

EXERCISES

6.5.1: Write functions to (a) insert x and (b) delete x from a list L, using linear search of an array.

6.5.2: Repeat Exercise 6.5.1 for an array with sentinels.

6.5.3: Repeat Exercise 6.5.1 for a sorted array.

6.5.4: Write the following functions assuming that list elements are of some arbitrary type `ETYPE`, for which we have functions $eq(x, y)$ that tells whether x and y are equal and $lt(x, y)$ telling whether x precedes y in the order of elements of type `ETYPE`.

a) Function *lookup* of Fig. 6.11.
b) Function *lookup* of Fig. 6.12.
c) Function *binsearch* of Fig. 6.14.

Probes in binary search

6.5.5**: Let $P(k)$ be the length $(high - low + 1)$ of the longest array such that the binary search algorithm of Fig. 6.14 never makes more than k *probes* [evaluations of `mid` at line (3)]. For example, $P(1) = 1$, and $P(2) = 3$. Write a recurrence relation for $P(k)$. What is the solution to your recurrence relation? Does it demonstrate that binary search makes $O(\log n)$ probes?

6.5.6*: Prove by induction on the difference between *low* and *high* that if x is in the range `A[low..high]`, then the binary search algorithm of Fig. 6.14 will find x.

6.5.7: Suppose we allowed arrays to have duplicates, so insertion could be done in $O(1)$ time. Write *insert*, *delete*, and *lookup* functions for this data structure.

6.5.8: Rewrite the binary search program to use iteration rather than recursion.

6.5.9**: Set up and solve a recurrence relation for the running time of binary search on an array of n elements. *Hint*: To simplify, it helps to take $T(n)$ as an upper bound on the running time of binary search on any array of n or fewer elements (rather than on exactly n elements, as would be our usual approach).

Ternary search

6.5.10: In *ternary search*, given a range *low* to *high*, we compute the approximate 1/3 point of the range,

$$first = \lfloor (2 \times low + high)/3 \rfloor$$

and compare the lookup element x with $A[first]$. If $x > A[first]$, we compute the approximate 2/3 point,

$$second = \lceil (low + 2 \times high)/3 \rceil$$

and compare x with $A[second]$. Thus we isolate x to within one of three ranges, each no more than one third the range *low* to *high*. Write a function to perform ternary search.

6.5.11**: Repeat Exercise 6.5.5 for ternary search. That is, find and solve a recurrence relation for the largest array that requires no more than k probes during ternary search. How do the number of probes required for binary and ternary search compare? That is, for a given k, can we handle larger arrays by binary search or by ternary search?

6.6 Stacks

Top of stack

A *stack* is an abstract data type based on the list data model in which all operations are performed at one end of the list, which is called the *top* of the stack. The term "LIFO (for last-in first-out) list" is a synonym for stack.

The abstract model of a stack is the same as that of a list — that is, a sequence of elements $a_1, a_2, \ldots, a_n$ of some one type. What distinguishes stacks from general lists is the particular set of operations permitted. We shall give a more complete set of operations later, but for the moment, we note that the quintessential stack operations are *push* and *pop*, where $push(x)$ puts the element x on top of the stack and *pop* removes the topmost element from the stack. If we write stacks with the top at the right end, the operation $push(x)$ applied to the list $(a_1, a_2, \ldots, a_n)$ yields the list $(a_1, a_2, \ldots, a_n, x)$. Popping the list $(a_1, a_2, \ldots, a_n)$ yields the list $(a_1, a_2, \ldots, a_{n-1})$; popping the empty list, ϵ, is impossible and causes an error condition,

Push and pop

✦ **Example 6.9.** Many compilers begin by turning the infix expressions that appear in programs into equivalent postfix expressions. For example, the expression $(3+4) \times 2$ is $3\ 4 + 2\ \times$ in postfix notation. A stack can be used to evaluate postfix expressions. Starting with an empty stack, we scan the postfix expression from left to right. Each time we encounter an argument, we push it onto the stack. When we encounter an operator, we pop the stack twice, remembering the operands popped. We then apply the operator to the two popped values (with the second as the left operand) and push the result onto the stack. Figure 6.15 shows the stack after each step in the processing of the postfix expression $3\ 4 + 2\ \times$. The result, 14, remains on the stack after processing. ✦

SYMBOL PROCESSED	STACK	ACTIONS
initial	ϵ	
3	3	*push* 3
4	3, 4	*push* 4
$+$	ϵ	*pop* 4; *pop* 3
		compute $7 = 3 + 4$
	7	*push* 7
2	7, 2	*push* 2
$\times$	ϵ	*pop* 2; *pop* 7
		compute $14 = 7 \times 2$
	14	*push* 14

Fig. 6.15. Evaluating a postfix expression using a stack.

Operations on a Stack

The ADT stack

The two previous ADT's we discussed, the dictionary and the priority queue, each had a definite set of associated operations. The stack is really a family of similar ADT's with the same underlying model, but with some variation in the set of allowable operations. In this section, we shall discuss the common operations on stacks and show two data structures that can serve as implementations for the stack, one based on linked lists and the other on arrays.

In any collection of stack operations we expect to see *push* and *pop*, as we mentioned. There is another common thread to the operations chosen for the stack ADT(s): they can all be implemented simply in $O(1)$ time, independent of the number of elements on the stack. You can check as an exercise that for the two data structures suggested, all operations require only constant time.

Clear stack

In addition to *push* and *pop*, we generally need an operation *clear* that initializes the stack to be empty. In Example 6.9, we tacitly assumed that the stack started out empty, without explaining how it got that way. Another possible operation is a test to determine whether the stack is currently empty.

Full and empty stacks

The last of the operations we shall consider is a test whether the stack is "full." Now in our abstract model of a stack, there is no notion of a full stack, since a stack is a list and lists can grow as long as we like, in principle. However, in any implementation of a stack, there will be some length beyond which it cannot grow. The most common example is when we represent a list or stack by an array. As seen in the previous section, we had to assume the list would not grow beyond the constant `MAX`, or our implementation of *insert* would not work.

The formal definitions of the operations we shall use in our implementation of stacks are the following. Let S be a stack of type `ETYPE` and x an element of type `ETYPE`.

1. $clear(S)$. Make the stack S empty.
2. $isEmpty(S)$. Return `TRUE` if S is empty, `FALSE` otherwise.
3. $isFull(S)$. Return `TRUE` if S is full, `FALSE` otherwise.
4. $pop(S, x)$. If S is empty, return `FALSE`; otherwise, set x to the value of the top element on stack S, remove this element from S, and return `TRUE`.
5. $push(x, S)$. If S is full, return `FALSE`; otherwise, add the element x to the top of S and return `TRUE`.

There is a common variation of *pop* that assumes S is nonempty. It takes only S as an argument and returns the element x that is popped. Yet another alternative version of *pop* does not return a value at all; it just removes the element at the top of the stack. Similarly, we may write *push* with the assumption that S is not "full." In that case, *push* does not return any value.

Array Implementation of Stacks

The implementations we used for lists can also be used for stacks. We shall discuss an array-based implementation first, followed by a linked-list representation. In each case, we take the type of elements to be `int`. Generalizations are left as exercises.

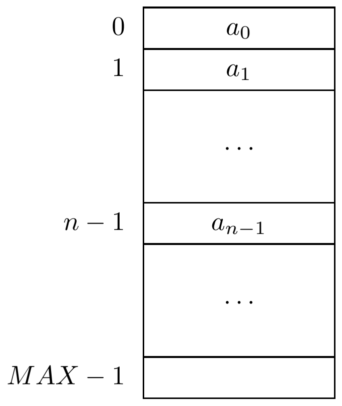


Fig. 6.16. An array representing a stack.

The declaration for an array-based stack of integers is

```
typedef struct {
    int A[MAX];
    int top;
} STACK;
```

```
void clear(STACK *pS)
{
    pS->top = -1;
}

BOOLEAN isEmpty(STACK *pS)
{
    return (pS->top < 0);
}

BOOLEAN isFull(STACK *pS)
{
    return (pS->top >= MAX-1);
}

BOOLEAN pop(STACK *pS, int *px)
{
    if (isEmpty(pS))
        return FALSE;
    else {
        (*px) = pS->A[(pS->top)--];
        return TRUE;
    }
}

BOOLEAN push(int x, STACK *pS)
{
    if (isFull(pS))
        return FALSE;
    else {
        pS->A[++(pS->top)] = x;
        return TRUE;
    }
}
```

Fig. 6.17. Functions to implement stack operations on arrays.

With an array-based implementation, the stack can grow either upward (from lower locations to higher) or downward (from higher locations to lower). We choose to have the stack grow upward;[5] that is, the oldest element a_0 in the stack is in location 0, the next-to-oldest element a_1 is in location 1, and the most recently inserted element a_{n-1} is in the location $n-1$.

The field `top` in the array structure indicates the position of the top of stack. Thus, in Fig. 6.16, `top` has the value $n-1$. An empty stack is represented by having $top = -1$. In that case, the content of array `A` is irrelevant, there being no elements on the stack.

The programs for the five stack operations defined earlier in this section are

[5] Thus the "top" of the stack is physically shown at the bottom of the page, an unfortunate but quite standard convention.

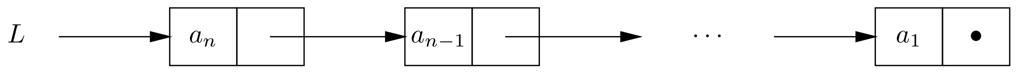
```
void clear(STACK *pS)
{
    (*pS) = NULL;
}

BOOLEAN isEmpty(STACK *pS)
{
    return ((*pS) == NULL);
}

BOOLEAN isFull(STACK *pS)
{
    return FALSE;
}

BOOLEAN pop(STACK *pS, int *px)
{
    if ((*pS) == NULL)
        return FALSE;
    else {
        (*px) = (*pS)->element;
        (*pS) = (*pS)->next;
        return TRUE;
    }
}

BOOLEAN push(int x, STACK *pS)
{
    STACK newCell;

    newCell = (STACK) malloc(sizeof(struct CELL));
    newCell->element = x;
    newCell->next = (*pS);
    (*pS) = newCell;
    return TRUE;
}
```

Fig. 6.18. Functions to implement stacks by linked lists.

shown in Fig. 6.17. We pass stacks by reference to avoid having to copy large arrays that are arguments of the functions.

Linked-List Implementation of a Stack

We can represent a stack by a linked-list data structure, like any list. However, it is convenient if the top of the stack is the front of the list. That way, we can push and pop at the head of the list, which takes only $O(1)$ time. If we had to find the end of the list to push or pop, it would take $O(n)$ time to do these operations on a stack of length n. However, as a consequence, the stack $S = (a_1, a_2, \ldots, a_n)$ must be represented "backward" by the linked list, as:



The type definition macro we have used for list cells can as well be used for stacks. The macro

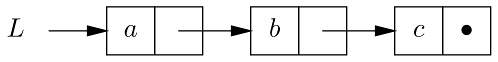
```
DefCell(int, CELL, STACK);
```

defines stacks of integers, expanding into

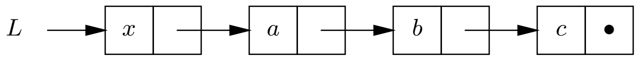
```
typdef struct CELL *STACK;
struct CELL {
    int element;
    STACK next;
};
```

With this representation, the five operations can be implemented by the functions in Fig. 6.18. We assume that `malloc` never runs out of space, which means that the *isFull* operation always returns `FALSE`, and the *push* operation never fails.

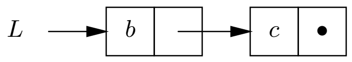
The effects of *push* and *pop* on a stack implemented as a linked list are illustrated in Fig. 6.19.



(a) List L.



(b) After executing $push(x, L)$.



(c) After executing $pop(L, x)$ on list L of (a).

Fig. 6.19. Push and pop operations on a stack implemented as a linked list.

EXERCISES

6.6.1: Show the stack that remains after executing the following sequence of operations, starting with an empty stack: $push(a)$, $push(b)$, *pop*, $push(c)$, $push(d)$, *pop*, $push(e)$, *pop*, *pop*.

6.6.2: Using only the five operations on stacks discussed in this section to manipulate the stack, write a C program to evaluate postfix expressions with integer operands and the four usual arithmetic operators, following the algorithm suggested in Example 6.9. Show that you can use either the array or the linked-list implementation with your program by defining the data type `STACK` appropriately and including with your program first the functions of Fig. 6.17, and then the functions of Fig. 6.18.

6.6.3*: How would you use a stack to evaluate prefix expressions?

6.6.4: Compute the running time of each of the functions in Figs. 6.17 and 6.18. Are they all $O(1)$?

6.6.5: Sometimes, a stack ADT uses an operation *top*, where *top*(S) returns the top element of stack S, which must be assumed nonempty. Write functions for *top* that can be used with

a) The array data structure
b) The linked-list data structure

that we defined for stacks in this section. Do your implementations of *top* all take $O(1)$ time?

6.6.6: Simulate a stack evaluating the following postfix expressions.

a) $ab + cd \times +e\times$
b) $abcde + + + +$
c) $ab + c + d + e+$

6.6.7*: Suppose we start with an empty stack and perform some push and pop operations. If the stack after these operations is $(a_1, a_2, \ldots, a_n)$ (top at the right), prove that a_i was pushed before a_{i+1} was pushed, for $i = 1, 2, \ldots, n-1$.

6.7 Implementing Function Calls Using a Stack

An important application of stacks is normally hidden from view: a stack is used to allocate space in the computer's memory to the variables belonging to the various functions of a program. We shall discuss the mechanism used in C, although a similar mechanism is used in almost every other programming language as well.

```
          int fact(int n)
          {
(1)           if (n <= 1)
(2)               return 1; /* basis */
              else
(3)               return n*fact(n-1); /* induction */
          }
```

Fig. 6.20. Recursive function to compute $n!$.

To see what the problem is, consider the simple, recursive factorial function `fact` from Section 2.7, which we reproduce here as Fig. 6.20. The function has a parameter `n` and a return value. As `fact` calls itself recursively, different calls are active at the same time. These calls have different values of the parameter `n` and produce different return values. Where are these different objects with the same names kept?

Run-time organization

To answer the question, we must learn a little about the *run-time organization* associated with a programming language. The run-time organization is the plan used to subdivide the computer's memory into regions to hold the various data items used by a program. When a program is run, each execution of a function is called an *activation.* The data objects associated with each activation are stored in the memory of the computer in a block called an *activation record* for that activation. The data objects include parameters, the return value, the return address, and any variables local to the function.

Activation record

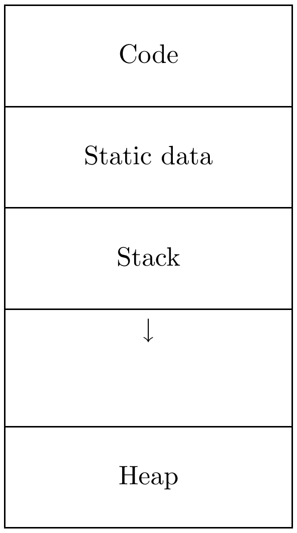


Fig. 6.21. Typical run-time memory organization.

Static data
Run-time stack

Figure 6.21 shows a typical subdivision of run-time memory. The first area contains the object code for the program being executed. The next area contains the static data for the program, such as the values of certain constants and external variables used by the program. The third area is the *run-time stack,* which grows downward toward the higher addresses in memory. At the highest-numbered memory locations is the *heap*, an area set aside for the objects that are dynamically allocated using `malloc`.[6]

The run-time stack holds the activation records for all the currently live activations. A stack is the appropriate structure, because when we call a function, we can push an activation record onto the stack. At all times, the currently executing activation A_1 has its activation record at the top of the stack. Just below the top of the stack is the activation record for the activation A_2 that called A_1. Below A_2's activation record is the record for the activation that called A_2, and so on. When a function returns, we pop its activation record off the top of stack, exposing the activation record of the function that called it. That is exactly the right thing to do, because when a function returns, control passes to the calling function.

✦ **Example 6.10.** Consider the skeletal program shown in Fig. 6.22. This program is nonrecursive, and there is never more than one activation for any one function.

[6] Do not confuse this use of the term "heap" with the heap data structure discussed in Section 5.9.

```
void P();
void Q();

main() {
    int x, y, z;

    P(); /* Here */
}

void P();
{
    int p1, p2;

    Q();
}

void Q()
{
    int q1, q2, q3;
            ...
}
```

Fig. 6.22. Skeletal program.

When the main function starts to execute, its activation record containing the space for the variables `x`, `y`, and `z` is pushed onto the stack. When function `P` is called, at the place marked `Here`, its activation record, which contains the space for the variables `p1` and `p2`, is pushed onto the stack.[7] When `P` calls `Q`, `Q`'s activation record is pushed onto the stack. At this point, the stack is as shown in Fig. 6.23.

When `Q` finishes executing, its activation record is popped off the stack. At that time, `P` is also finished, and so its activation record is popped. Finally, `main` too is finished and has its activation record popped off the stack. Now the stack is empty, and the program is finished. ✦

✦ **Example 6.11.** Consider the recursive function `fact` from Fig. 6.20. There may be many activations of `fact` live at any one time, but each one will have an activation record of the same form, namely

n
fact

consisting of a word for the parameter `n`, which is filled initially, and a word for the return value, which we have denoted `fact`. The return value is not filled until the last step of the activation, just before the return.

[7] Notice that the activation record for P has two data objects, and so is of a "type" different from that of the activation record for the main program. However, we may regard all activation record forms for a program as variants of a single record type, thus preserving the viewpoint that a stack has all its elements of the same type.

x
y
z
p1
p2
q1
q2
q3

Fig. 6.23. Run-time stack when function `Q` is executing.

Suppose we call `fact(4)`. Then we create one activation record, of the form

n	4
fact	-

As `fact(4)` calls `fact(3)`, we next push an activation record for that activation onto the run-time stack, which now appears as

n	4
fact	-
n	3
fact	-

Note that there are two locations named `n` and two named `fact`. There is no confusion, since they belong to different activations and only one activation record can be at the top of the stack at any one time: the activation record belonging to the currently executing activation.

n	4
fact	-
n	3
fact	-
n	2
fact	-
n	1
fact	-

Fig. 6.24. Activation records during execution of `fact`.

Then `fact(3)` calls `fact(2)`, which calls `fact(1)`. At that point, the run-time stack is as in Fig. 6.24. Now `fact(1)` makes no recursive call, but assigns $fact = 1$. The value 1 is thus placed in the slot of the top activation record reserved for `fact`.

n	4
fact	-
n	3
fact	-
n	2
fact	-
n	1
fact	1

Fig. 6.25. After `fact(1)` computes its value.

The other slots labeled `fact` are unaffected, as shown in Fig. 6.25.

Then, `fact(1)` returns, exposing the activation record for `fact(2)` and returning control to the activation `fact(2)` at the point where `fact(1)` was called. The return value, 1, from `fact(1)` is multiplied by the value of n in the activation record for `fact(2)`, and the product is placed in the slot for `fact` of that activation record, as required by line (3) in Fig. 6.20. The resulting stack is shown in Fig. 6.26.

n	4
fact	-
n	3
fact	-
n	2
fact	2

Fig. 6.26. After `fact(2)` computes its value.

Similarly, `fact(2)` then returns control to `fact(3)`, and the activation record for `fact(2)` is popped off the stack. The return value, 2, multiplies n of `fact(3)`, producing the return value 6. Then, `fact(3)` returns, and its return value multiplies `n` in `fact(4)`, producing the return value 24. The stack is now

n	4
fact	24

At this point, `fact(4)` returns to some hypothetical calling function whose activation record (not shown) is below that of `fact(4)` on the stack. However, it would receive the return value 24 as the value of `fact(4)`, and would proceed with its own execution. ✦

EXERCISES

6.7.1: Consider the C program of Fig. 6.27. The activation record for `main` has a slot for the integer `i`. The important data in the activation record for `sum` is

```
     #define MAX 4
     int A[MAX];
     int sum(int i);

     main()
     {
         int i;

(1)      for (i = 0; i < MAX; i++)
(2)          scanf("%d", &A[i]);
(3)      printf("\n", sum(0));
     }

     int sum(int i)
     {
(4)          if (i > MAX)
(5)              return 0;
             else
(6)              return A[i] + sum(i+1);
     }
```

Fig. 6.27. Program for Exercise 6.7.1.

1. The parameter `i`.
2. The return value.
3. An unnamed temporary location, which we shall call `temp`, to store the value of `sum(i+1)`. The latter is computed in line (6) and then added to $A[i]$ to form the return value.

Show the stack of activation records immediately before and immediately after each call to `sum`, on the assumption that the value of $A[i]$ is $10i$. That is, show the stack immediately after we have pushed an activation record for `sum`, and just before we pop an activation record off the stack. You need not show the contents of the bottom activation record (for `main`) each time.

```
void delete(int x, LIST *pL)
{
    if ((*pL) != NULL)
        if (x == (*pL)->element)
            (*pL) = (*pL)->next;
        else
            delete(x, &((*pL)->next));
end
```

Fig. 6.28. Function for Exercise 6.7.2.

6.7.2*: The function `delete` of Fig. 6.28 removes the first occurrence of integer x from a linked list composed of the usual cells defined by

```
    DefCell(int, CELL, LIST);
```

The activation record for `delete` consists of the parameters `x` and `pL`. However, since `pL` is a pointer to a list, the value of the second parameter in the activation record is not a pointer to the first cell on the list, but rather a pointer to a pointer to the first cell. Typically, an activation record will hold a pointer to the `next` field of some cell. Show the sequence of stacks when `delete(3,&L)` is called (from some other function) and L is a pointer to the first cell of a linked list containing elements 1, 2, 3, and 4, in that order.

6.8 Queues

Front and rear of queue

Another important ADT based on the list data model is the *queue*, a restricted form of list in which elements are inserted at one end, the *rear*, and removed from the other end, the *front*. The term "FIFO (first-in first-out) list" is a synonym for queue.

The intuitive idea behind a queue is a line at a cashier's window. People enter the line at the rear and receive service once they reach the front. Unlike a stack, there is fairness to a queue; people are served in the order in which they enter the line. Thus the person who has waited the longest is the one who is served next.

Operations on a Queue

Enqueue and dequeue

The abstract model of a queue is the same as that of a list (or a stack), but the operations applied are special. The two operations that are characteristic of a queue are *enqueue* and *dequeue*; *enqueue*(x) adds x to the rear of a queue, *dequeue* removes the element from the front of the queue. As is true of stacks, there are certain other useful operations that we may want to apply to queues.

Let Q be a queue whose elements are of type `ETYPE`, and let x be an element of type `ETYPE`. We shall consider the following operations on queues:

1. *clear*(Q). Make the queue Q empty.
2. *dequeue*(Q, x). If Q is empty, return `FALSE`; otherwise, set x to the value of the element at the front of Q, remove this element from Q, and return `TRUE`.
3. *enqueue*(x, Q). If Q is full, return `FALSE`; otherwise, add the element x to the rear of Q and return `TRUE`.
4. *isEmpty*(Q). Return `TRUE` if Q is empty and `FALSE` otherwise.
5. *isFull*(Q). Return `TRUE` if Q is full and `FALSE` otherwise.

As with stacks, we can have more "trusting" versions of *enqueue* and *dequeue* that do not check for a full or empty queue, respectively. Then *enqueue* does not return a value, and *dequeue* takes only Q as an argument and returns the value dequeued.

A Linked-List Implementation of Queues

A useful data structure for queues is based on linked lists. We start with the usual definition of cells given by the macro

```
void clear(QUEUE *pQ)
{
    pQ->front = NULL;
}

BOOLEAN isEmpty(QUEUE *pQ)
{
    return (pQ->front == NULL);
}

BOOLEAN isFull(QUEUE *pQ)
{
    return FALSE;
}

BOOLEAN dequeue(QUEUE *pQ, int *px)
{
    if (isEmpty(pQ))
        return FALSE;
    else {
        (*px) = pQ->front->element;
        pQ->front = pQ->front->next;
        return TRUE;
    }
}

BOOLEAN enqueue(int x, QUEUE *pQ)
{
    if (isEmpty(pQ)) {
        pQ->front = (LIST) malloc(sizeof(struct CELL));
        pQ->rear = pQ->front;
    }
    else {
        pQ->rear->next = (LIST) malloc(sizeof(struct CELL));
        pQ->rear = pQ->rear->next;
    }
    pQ->rear->element = x;
    pQ->rear->next = NULL;
    return TRUE;
}
```

Fig. 6.29. Procedures to implement linked-list queue operations.

```
DefCell(int, CELL, LIST);
```

As previously in this chapter, we assume that elements of our queues are integers and invite the reader to generalize our functions to arbitrary element types.

The elements of a queue will be stored on a linked list of cells. A queue itself is a structure with two pointers — one to the front cell (the first on the linked list) and another to the rear cell (the last on the linked list). That is, we define

More Abstract Data Types

We can add the stack and queue to the table of ADT's that we started in Section 5.9. We covered two data structures for the stack in Section 6.6, and one data structure for the queue in Section 6.8. Exercise 6.8.3 covers another data structure, the "circular array," for the queue.

ADT	Stack	Queue
Abstract Implementation	List	List
Data Structures	1) Linked List 2) Array	1) Linked List 2) Circular Array

```
    typedef struct {
        LIST front, rear;
    } QUEUE;
```

If the queue is empty, `front` will be `NULL`, and the value of `rear` is then irrelevant.

Figure 6.29 gives programs for the queue operations mentioned in this section. Note that when a linked list is used there is no notion of a "full" queue, and so *isFull* returns `FALSE` always. However, if we used some sort of array-based implementation of a queue, there would be the possibility of a full queue.

EXERCISES

6.8.1: Show the queue that remains after executing the following sequence of operations, starting with an empty queue: *enqueue*(a), *enqueue*(b), *dequeue*, *enqueue*(c), *enqueue*(d), *dequeue*, *enqueue*(e), *dequeue*, *dequeue*.

6.8.2: Show that each of the functions in Fig. 6.29 can be executed in $O(1)$ time, regardless of the length of the queue.

Circular array

6.8.3*: We can represent a queue by an array, provided that the queue does not grow too long. In order to make the operations take $O(1)$ time, we must think of the array as *circular*. That is, the array `A[0..n-1]` can be thought of as having `A[1]` follow `A[0]`, `A[2]` follow `A[1]`, and so on, up to `A[n-1]` following `A[n-2]`, but also `A[0]` follows `A[n-1]`. The queue is represented by a pair of integers `front` and `rear` that indicate the positions of the front and rear elements of the queue. An empty queue is represented by having $front$ be the position that follows $rear$ in the circular sense; for example, $front = 23$ and $rear = 22$, or $front = 0$ and $rear = n - 1$. Note that therefore, the queue cannot have n elements, or that condition too would be represented by $front$ immediately following $rear$. Thus the queue is full when it has $n-1$ elements, not when it has n elements. Write functions for the queue operations assuming the circular array data structure. Do not forget to check for full and empty queues.

6.8.4*: Show that if $(a_1, a_2, \ldots, a_n)$ is a queue with a_1 at the front, then a_i was enqueued before a_{i+1}, for $i = 1, 2, \ldots, n-1$.

✦✦✦ 6.9 Longest Common Subsequences

This section is devoted to an interesting problem about lists. Suppose we have two lists and we want to know what differences there are between them. This problem appears in many different guises; perhaps the most common occurs when the two lists represent two different versions of a text file and we want to determine which lines are common to the two versions. For notational convenience, throughout this section we shall assume that lists are character strings.

A useful way to think about this problem is to treat the two files as sequences of symbols, $x = a_1 \cdots a_m$ and $y = b_1 \cdots b_n$, where a_i represents the ith line of the first file and b_j represents the jth line of the second file. Thus an abstract symbol like a_i may really be a "big" object, perhaps a full sentence.

Diff command

There is a UNIX command `diff` that compares two text files for their differences. One file, x, might be the current version of a program and the other, y, might be the version of the program before a small change was made. We could use `diff` to remind ourselves of the changes that were made turning y into x. The typical changes that are made to a text file are

1. Inserting a line.
2. Deleting a line.

A modification of a line can be treated as a deletion followed by an insertion.

Usually, if we examine two text files in which a small number of such changes have been made when transforming one into the other, it is easy to see which lines correspond to which, and which lines have been deleted and which inserted. The `diff` command makes the assumption that one can identify what the changes are by first finding a *longest common subsequence,* or *LCS*, of the two lists whose elements are the lines of the two text files involved. An LCS represents those lines that have not been changed.

LCS

Recall that a subsequence is formed from a list by deleting zero or more elements, keeping the remaining elements in order. A *common subsequence* of two lists is a list that is a subsequence of both. A longest common subsequence of two lists is a common subsequence that is as long as any common subsequence of the two lists.

Common subsequence

✦ **Example 6.12.** In what follows, we can think of characters like `a`, `b`, or `c`, as standing for lines of a text file, or as any other type of elements if we wish. As an example, `baba` and `cbba` are both longest common subsequences of `abcabba` and `cbabac`. We see that `baba` is a subsequence of `abcabba`, because we may take positions 2, 4, 5, and 7 of the latter string to form `baba`. String `baba` is also a subsequence of `cbabac`, because we may take positions 2, 3, 4, and 5. Similarly, `cbba` is formed from positions 3, 5, 6, and 7 of `abcabba` and from positions 1, 2, 4, and 5 of `cbabac`. Thus `cbba` too is a common subsequence of these strings. We must convince ourselves that these are longest common subsequences; that is, there are no common subsequences of length five or more. That fact will follow from the algorithm we describe next. ✦

A Recursion That Computes the LCS

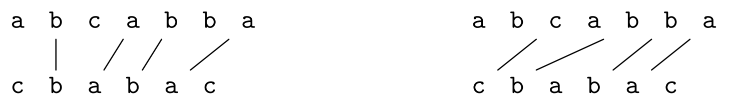
We offer a recursive definition of the length of the LCS of two lists. This definition will let us calculate the length easily, and by examining the table it constructs, we can then discover one of the possible LCS's itself, rather than just its length. From the LCS, we can deduce what changes were made to the text files in question; essentially, everything that is not part of the LCS is a change.

To find the length of an LCS of lists x and y, we need to find the lengths of the LCS's of all pairs of prefixes, one from x and the other from y. Recall that a prefix is an initial sublist of a list, so that, for instance, the prefixes of `cbabac` are ϵ, `c`, `cb`, `cba`, and so on. Suppose that $x = (a_1, a_2, \ldots, a_m)$ and $y = (b_1, b_2, \ldots, b_n)$. For each i and j, where i is between 0 and m and j is between 0 and n, we can ask for an LCS of the prefix $(a_1, \ldots, a_i)$ from x and the prefix $(b_1, \ldots, b_j)$ from y.

If either i or j is 0, then one of the prefixes is ϵ, and the only possible common subsequence of the two prefixes is ϵ. Thus when either i or j is 0, the length of the LCS is 0. This observation is formalized in both the basis and rule (1) of the induction that follows our informal discussion of how the LCS is computed.

Now consider the case where both i and j are greater than 0. It helps to think of an LCS as a matching between certain positions of the two strings involved. That is, for each element of the LCS, we match the two positions of the two strings from which that element comes. Matched positions must have the same symbols, and the lines between matched positions must not cross.

✦ **Example 6.13.** Figure 6.30(a) shows one of two possible matchings between strings `abcabba` and `cbabac` corresponding to the common subsequence `baba` and Fig. 6.30(b) shows a matching corresponding to `cbba`. ✦



(a) For `baba`. (b) For `cbba`.

Fig. 6.30. LCS's as matchings between positions.

Thus let us consider any matching between prefixes $(a_1, \ldots, a_i)$ and $(b_1, \ldots, b_j)$. There are two cases, depending on whether or not the last symbols of the two lists are equal.

a) If $a_i \neq b_j$, then the matching cannot include both a_i and b_j. Thus an LCS of $(a_1, \ldots, a_i)$ and $(b_1, \ldots, b_j)$ must be either

   i) An LCS of $(a_1, \ldots, a_{i-1})$ and $(b_1, \ldots, b_j)$, or

   ii) An LCS of $(a_1, \ldots, a_i)$ and $(b_1, \ldots, b_{j-1})$.

If we have already found the lengths of the LCS's of these two pairs of prefixes, then we can take the larger to be the length of the LCS of $(a_1, \ldots, a_i)$ and $(b_1, \ldots, b_j)$. This situation is formalized in rule (2) of the induction that follows.

b) If $a_i = b_j$, we can match a_i and b_j, and the matching will not interfere with any other potential matches. Thus the length of the LCS of $(a_1, \ldots, a_i)$ and $(b_1, \ldots, b_j)$ is 1 greater than the length of the LCS of $(a_1, \ldots, a_{i-1})$ and $(b_1, \ldots, b_{j-1})$. This situation is formalized in rule (3) of the following induction.

These observations let us give a recursive definition for $L(i, j)$, the length of the LCS of $(a_1, \ldots, a_i)$ and $(b_1, \ldots, b_j)$. We use complete induction on the sum $i + j$.

BASIS. If $i + j = 0$, then both i and j are 0, and so the LCS is ϵ. Thus $L(0, 0) = 0$.

INDUCTION. Consider i and j, and suppose we have already computed $L(g, h)$ for any g and h such that $g + h < i + j$. There are three cases to consider.

1. If either i or j is 0, then $L(i, j) = 0$.
2. If $i > 0$ and $j > 0$, and $a_i \neq b_j$, then $L(i, j) = \max\bigl(L(i, j-1), L(i-1, j)\bigr)$.
3. If $i > 0$ and $j > 0$, and $a_i = b_j$, then $L(i, j) = 1 + L(i-1, j-1)$.

A Dynamic Programming Algorithm for the LCS

Ultimately what we want is $L(m, n)$, the length of an LCS for the two lists x and y. If we write a recursive program based on the preceding induction, it will take time that is exponential in the smaller of m and n. That is far too much time to make the simple recursive algorithm practical for, say, $n = m = 100$. The reason this recursion does so badly is a bit subtle. To begin, suppose there are no matches at all between characters in the lists x and y, and we call $L(3, 3)$. That results in calls to $L(2, 3)$ and $L(3, 2)$. But each of these calls results in a call to $L(2, 2)$. We thus do the work of $L(2, 2)$ twice. The number of times $L(i, j)$ is called increases rapidly as the arguments of L become smaller. If we continue the trace of calls, we find that $L(1, 1)$ is called 6 times, $L(0, 1)$ and $L(1, 0)$ are called 10 times each, and $L(0, 0)$ is called 20 times.

We can do much better if we build a two-dimensional table, or array, to store $L(i, j)$ for the various values of i and j. If we compute the values in order of the induction — that is, smallest values of $i + j$ first — then the needed values of L are always in the table when we compute $L(i, j)$. In fact, it is easier to compute L by rows, that is, for $i = 0, 1, 2$, and so on; within a row, compute by columns, for $j = 0, 1, 2$, and so on. Again, we can be sure of finding the needed values in the table when we compute $L(i, j)$, and no recursive calls are necessary. As a result, it takes only $O(1)$ time to compute each entry of the table, and a table for the LCS of lists of length m and n can be constructed in $O(mn)$ time.

In Fig. 6.31 we see C code that fills this table, working by row rather than by the sum $i + j$. We assume that the list x is stored in an array `a[1..m]` and y is stored in `b[1..n]`. Note that the 0th elements of these are unused; doing so simplifies the notation in Fig. 6.31. We leave it as an exercise to show that the running time of this program is $O(mn)$ on lists of length m and n.[8]

[8] Strictly speaking, we discussed only big-oh expressions that are a function of one variable. However, the meaning here should be clear. If $T(m, n)$ is the running time of the program

```
for (j = 0; j <= n; j++)
    L[0][j] = 0;
for (i = 1; i <= m; i++) {
    L[i][0] = 0;
    for (j = 1; j <= n; j++)
        if (a[i] != b[j])
            if (L[i-1][j] >= L[i][j-1])
                L[i][j] = L[i-1][j];
            else
                L[i][j] = L[i][j-1];
        else /* a[i] == b[j] */
            L[i][j] = 1 + L[i-1][j-1];
}
```

Fig. 6.31. C fragment to fill the LCS table.

Dynamic Programming

The term "dynamic programming" comes from a general theory developed by R. E. Bellman in the 1950's for solving problems in control systems. People who work in the field of artificial intelligence often speak of the technique under the name *memoing* or *tabulation.*

Memoing

A table-filling technique like this example is often called a *dynamic programming algorithm.* As in this case, it can be much more efficient than a straightforward implementation of a recursion that solves the same subproblem repeatedly.

Dynamic programming algorithm

✦ **Example 6.14.** Let x be the list `cbabac` and y the list `abcabba`. Figure 6.32 shows the table constructed for these two lists. For instance, $L(6,7)$ is a case where $a_6 \neq b_7$. Thus $L(6,7)$ is the larger of the entries just below and just to the left. Since these are 4 and 3, respectively, we set $L(6,7)$, the entry in the upper right corner, to 4. Now consider $L(4,5)$. Since both a_4 and b_5 are the symbol `b`, we add 1 to the entry $L(3,4)$ that we find to the lower left. Since that entry is 2, we set $L(4,5)$ to 3. ✦

Recovery of an LCS

We now have a table giving us the length of the LCS, not only for the lists in question, but for each pair of their prefixes. From this information we must deduce one of the possible LCS's for the two lists in question. To do so, we shall find the matching pairs of elements that form one of the LCS's. We shall find a path through the table, beginning at the upper right corner; this path will identify an LCS.

Suppose that our path, starting at the upper right corner, has taken us to row i and column j, the point in the table that corresponds to the pair of elements a_i

on lists of length m and n, then there are constants m_0, n_0, and c such that for all $m \geq m_0$ and $n \geq n_0$, $T(m,n) \leq cmn$.

c	6	0	1	2	3	3	3	3	4
a	5	0	1	2	2	3	3	3	4
b	4	0	1	2	2	2	3	3	3
a	3	0	1	1	1	2	2	2	3
b	2	0	0	1	1	1	2	2	2
c	1	0	0	0	1	1	1	1	1
	0	0	0	0	0	0	0	0	0
		0	1	2	3	4	5	6	7
			a	b	c	a	b	b	a

Fig. 6.32. Table of longest common subsequences for `cbabac` and `abcabba`.

and b_j. If $a_i = b_j$, then $L(i,j)$ was chosen to be $1 + L(i-1, j-1)$. We thus treat a_i and b_j as a matched pair of elements, and we shall include the symbol that is a_i (and also b_j) in the LCS, ahead of all the elements of the LCS found so far. We then move our path down and to the left, that is, to row $i-1$ and column $j-1$.

However, it is also possible that $a_i \neq b_j$. If so, then $L(i,j)$ must equal at least one of $L(i-1,j)$ and $L(i,j-1)$. If $L(i,j) = L(i-1,j)$, we shall move our path one row down, and if not, we know that $L(i,j) = L(i,j-1)$, and we shall move our path one column left.

When we follow this rule, we eventually arrive at the lower left corner. At that point, we have selected a certain sequence of elements for our LCS, and the LCS itself is the list of these elements, in the reverse of the order in which they were selected.

c	6	0	1	2	3	3	3	3	**4**
a	5	0	1	2	2	3	3	3	**4**
b	4	0	1	2	2	2	3	**3**	3
a	3	0	1	1	1	2	**2**	2	3
b	2	0	0	1	1	1	**2**	2	2
c	1	0	0	0	**1**	**1**	1	1	1
	0	**0**	**0**	**0**	0	0	0	0	0
		0	1	2	3	4	5	6	7
			a	b	c	a	b	b	a

Fig. 6.33. A path that finds the LCS `caba`.

✦ **Example 6.15.** The table of Fig. 6.32 is shown again in Fig. 6.33, with a path shown in bold. We start with $L(6,7)$, which is 4. Since $a_6 \neq b_7$, we look immediately to the left and down to find the value 4, which must appear in at least one of these places. In this case, 4 appears only below, and so we go to $L(5,7)$. Now $a_5 = b_7$; both are `a`. Thus `a` is the last symbol of the LCS, and we move southwest, to $L(4,6)$.

Since a_4 and b_6 are both `b`, we include `b`, ahead of `a`, in the LCS being formed, and we again move southwest, to $L(3,5)$. Here, we find $a_3 \neq b_5$, but $L(3,5)$, which is 2, equals both the entry below and the entry to the left. We have elected in this situation to move down, so we next move to $L(2,5)$. There we find $a_2 = b_5 =$ `b`, and so we put a `b` ahead of the LCS being formed and move southwest to $L(1,4)$.

Since $a_1 \neq b_4$ and only the entry to the left has the same value (1) as $L(1,4)$, we move to $L(1,3)$. Now we have $a_1 = b_3 =$ `c`, and so we add `c` to the beginning of the LCS and move to $L(0,2)$. At this point, we have no choice but to move left to $L(0,1)$ and then $L(0,0)$, and we are done. The resulting LCS consists of the four characters we discovered, in the reverse order, or `cbba`. That happens to be one of the two LCS's we mentioned in Example 6.12. We can obtain other LCS's by choosing to go left instead of down when $L(i,j)$ equals both $L(i,j-1)$ and $L(i-1,j)$, and by choosing to go left or down when one of these equals $L(i,j)$, even in the situation when $a_i = b_j$ (i.e., by skipping certain matches in favor of matches farther to the left). ✦

We can prove that this path finding algorithm always finds an LCS. The statement that we prove by complete induction on the sum of the lengths of the lists is:

STATEMENT $S(k)$: If we find ourselves at row i and column j, where $i + j = k$, and if $L(i,j) = v$, then we subsequently discover v elements for our LCS.

BASIS. The basis is $k = 0$. If $i + j = 0$, then both i and j are 0. We have finished our path and find no more elements for the LCS. As we know $L(0,0) = 0$, the inductive hypothesis holds for $i + j = 0$.

INDUCTION. Assume the inductive hypothesis for sums k or less, and let $i + j = k + 1$. Suppose we are at $L(i,j)$, which has value v. If $a_i = b_j$, then we find one match and move to $L(i-1,j-1)$. Since the sum $(i-1)+(j-1)$ is less than $i+j$, the inductive hypothesis applies. Since $L(i-1,j-1)$ must be $v-1$, we know that we shall find $v-1$ more elements for our LCS, which, with the one element just found, will give us v elements. That observation proves the inductive hypothesis in this case.

The only other case is when $a_i \neq b_j$. Then, either $L(i-1,j)$ or $L(i,j-1)$, or both, must have the value v, and we move to one of these positions that does have the value v. Since the sum of the row and column is $i + j - 1$ in either case, the inductive hypothesis applies, and we conclude that we find v elements for the LCS. Again we can conclude that $S(k+1)$ is true. Since we have considered all cases, we are done and conclude that if we are at an entry $L(i,j)$, we always find $L(i,j)$ elements for our LCS.

EXERCISES

6.9.1: What is the length of the LCS of the lists

a) `banana` and `cabana`
b) `abaacbacab` and `bacabbcaba`

6.9.2*: Find all the LCS's of the pairs of lists from Exercise 6.9.1. *Hint*: After building the table from Exercise 6.9.1, trace backward from the upper right corner, following each choice in turn when you come to a point that could be explained in two or three different ways.

6.9.3**: Suppose we use the recursive algorithm for computing the LCS that we described first (instead of the table-filling program that we recommend). If we call $L(4, 4)$ with two lists having no symbols in common, how many calls to $L(1, 1)$ are made? *Hint*: Use a table-filling (dynamic programming) algorithm to compute a table giving the value of $L(i, j)$ for all i and j. Compare your result with Pascal's triangle from Section 4.5. What does this relationship suggest about a formula for the number of calls?

6.9.4**: Suppose we have two lists x and y, each of length n. For n below a certain size, there can be at most one string that is an LCS of x and y (although that string may occur in different positions of x and/or y). For example, if $n = 1$, then the LCS can only be ϵ, unless x and y are both the same symbol a, in which case a is the only LCS. What is the smallest value of n for which x and y can have two different LCS's?

6.9.5: Show that the program of Fig. 6.31 has running time $O(mn)$.

6.9.6: Write a C program to take a table, such as that computed by the program of Fig. 6.31, and find the positions, in each string, of one LCS. What is the running time of your program, if the table is m by n?

6.9.7: In the beginning of this section, we suggested that the length of an LCS and the size of the largest matching between positions of two strings were related.

a*) Prove by induction on k that if two strings have a common subsequence of length k, then they have a matching of length k.

b) Prove that if two strings have a matching of length k, then they have a common subsequence of length k.

c) Conclude from (a) and (b) that the lengths of the LCS and the greatest size of a matching are the same.

6.10 Representing Character Strings

Character strings are probably the most common form of list encountered in practice. There are a great many ways to represent strings, and some of these techniques are rarely appropriate for other kinds of lists. Therefore, we shall devote this section to the special issues regarding character strings.

First, we should realize that storing a single character string is rarely the whole problem. Often, we have a large number of character strings, each rather short. They may form a dictionary, meaning that we insert and delete strings from the population as time goes on, or they may be a *static* set of strings, unchanging over time. The following are two typical examples.

Concordance

1. A useful tool for studying texts is a *concordance,* a list of all the words used in the document and the places in which they occur. There will typically be tens of thousands of different words used in a large document, and each occurrence must be stored once. The set of words used is static; that is, once formed it does not change, except perhaps if there were errors in the original concordance.

2. The compiler that turns a C program into machine code must keep track of all the character strings that represent variables of the program. A large program may have hundreds or thousands of variable names, especially when we remember that two local variables named `i` that are declared in two functions are really two distinct variables. As the compiler scans the program, it finds new variable names and inserts them into the set of names. Once the compiler has finished compiling a function, the variables of that function are not available to subsequent functions, and so may be deleted.

In both of these examples, there will be many short character strings. Short words abound in English, and programmers like to use single letters such as `i` or `x` for variables. On the other hand, there is no limit on the length of words, either in English texts or in programs.

Character Strings in C

Null character

Character-string constants, as might appear in a C program, are stored as arrays of characters, followed by the special character `'\0'`, called the *null character*, whose value is 0. However, in applications such as the ones mentioned above, we need the facility to create and store new strings as a program runs. Thus, we need a data structure in which we can store arbitrary character strings. Some of the possibilities are:

Truncation

1. Use a fixed-length array to hold character strings. Strings shorter than the array are followed by a null character. Strings longer than the array cannot be stored in their entirety. They must be *truncated* by storing only their prefix of length equal to the length of the array.

2. A scheme similar to (1), but assume that every string, or prefix of a truncated string, is followed by the null character. This approach simplifies the reading of strings, but it reduces by one the number of string characters that can be stored.

3. A scheme similar to (1), but instead of following strings by a null character, use another integer *length* to indicate how long the string really is.

4. To avoid the restriction of a maximum string length, we can store the characters of the string as the elements of a linked list. Possibly, several characters can be stored in one cell.

5. We may create a large array of characters in which individual character strings are placed. A string is then represented by a pointer to a place in the array where the string begins. Strings may be terminated by a null character or they may have an associated length.

Fixed-Length Array Representations

Let us consider a structure of type (1) above, where strings are represented by fixed-length arrays. In the following example, we create structures that have a fixed-length array as one of their fields.

Example 6.16. Consider the data structure we might use to hold one entry in a concordance, that is, a single word and its associated information. We need to hold

1. The word itself.
2. The number of times the word appears.
3. A list of the lines of the document in which there are one or more occurrences of the word.

Thus we might use the following structure:

```
typedef struct {
    char word[MAX];
    int occurrences;
    LIST lines;
} WORDCELL;
```

Here, `MAX` is the maximum length of a word. All `WORDCELL` structures have an array called `word` of `MAX` bytes, no matter how short the word happens to be.

The field `occurrences` is a count of the number of times the word appears, and `lines` is a pointer to the beginning of a linked list of cells. These cells are of the conventional type defined by the macro

```
DefCell(int, CELL, LIST);
```

Each cell holds one integer, representing a line on which there are one or more occurrences of the word in question. Note that `occurrences` could be larger than the length of the list, if the word appeared several times on one line.

In Fig. 6.34 we see the structure for the word `earth` in the first chapter of Genesis. We assume `MAX` is at least 6. The complete list of line (verse) numbers is (1, 2, 10, 11, 12, 15, 17, 20, 22, 24, 25, 26, 28, 29, 30).

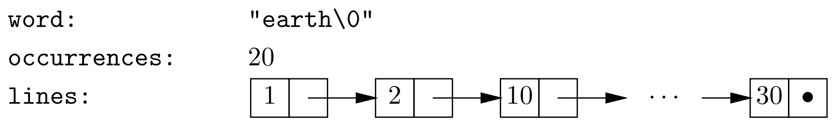


Fig. 6.34. Concordance entry for the word `earth` in the first chapter of Genesis.

The entire concordance might consist of a collection of structures of type `WORD-CELL`. These might, for example, be organized in a binary search tree, with the < ordering of structures based on the alphabetic order of words. That structure would allow relatively fast access to words as we use the concordance. It would also allow us to create the concordance efficiently as we scan the text to locate and list the occurrences of the various words. To use the binary tree structure we would require left- and right-child fields in the type `WORDCELL`. We could also arrange these structures in a linked list, by adding a "next" field to the type `WORDCELL` instead. That would be a simpler structure, but it would be less efficient if the number of words is large. We shall see, in the next chapter, how to arrange these structures in a hash table, which probably offers the best performance of all data structures for this problem. ✦

Linked Lists for Character Strings

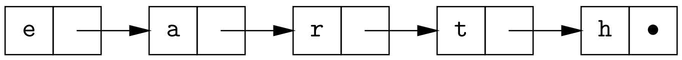
The limitation on the length of character strings, and the need to allocate a fixed amount of space no matter how short the string, are two disadvantages of the previous implementation of character strings. However, C and other languages allow us to build other, more flexible data structures to represent strings. For example, if we are concerned that there be no upper limit on the length of a character string, we can use conventional linked lists of characters to hold character strings. That is, we can declare a type

```
typedef struct CHARCELL *CHARSTRING;
struct CHARCELL {
    char character;
    CHARSTRING next;
};
```

In the type `WORDCELL`, `CHARSTRING` becomes the type of the field `word`, as

```
typedef {
    CHARSTRING word;
    int occurrences;
    LIST lines;
} WORDCELL;
```

For example, the word `earth` would be represented by



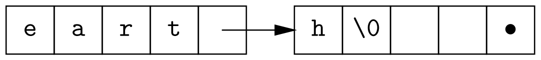
This scheme removes any upper limit on the length of words, but it is, in practice, not very economical of space. The reason is that each structure of type `CHARCELL` takes at least five bytes, assuming one for the character and a typical four for a pointer to the next cell on the list. Thus, the great majority of the space is used for the "overhead" of pointers rather than the "payload" of characters.

Packing characters into cells

We can be a bit more clever, however, if we pack several bytes into the data field of each cell. For example, if we put four characters into each cell, and pointers consume four bytes, then half our space will be used for "payload," compared with 20% payload in the one-character-per-cell scheme. The only caution is that we must have some character, such as the null character, that can serve as a string-terminating character, as is the case for character strings stored in arrays. In general, if `CPC` (characters per cell) is the number of characters that we are willing to place in one cell, we can declare cells by

```
typedef struct CHARCELL *CHARSTRING;
struct CHARCELL {
    char characters[CPC];
    CHARSTRING next;
};
```

For example, if CPC = 4, then we could store the word **earth** in two cells, as



We could also increase CPC above 4. As we do so, the fraction of space taken for pointers decreases, which is good; it means that the overhead of using linked lists rather than arrays is dropping. On the other hand, if we used a very large value for CPC, we would find that almost all words used only one cell, but that cell would have many unused locations in it, just as an array of length CPC would.

✦ **Example 6.17.** Let us suppose that in our population of character strings, 30% are between 1 and 4 characters long, 40% between 5 and 8 characters, 20% in the range 9–12, and 10% in the range 13–16. Then the table in Fig. 6.35 gives the number of bytes devoted to linked lists representing words in the four ranges, for four values of CPC, namely, 4, 8, 12, and 16. For our assumption about word-length frequencies, CPC = 8 comes out best, with an average usage of 15.6 bytes That is, we are best off using cells with room for 8 bytes, using a total of 12 bytes per cell, including the 4 bytes for the next pointer. Note that the total space cost, which is 19.6 bytes when we include a pointer to the front of the list, is not as good as using 16 bytes for a character array. However, the linked-list scheme can accommodate strings longer than 16 characters, even though our assumptions put a 0% probability on finding such strings. ✦

		CHARACTERS PER CELL			
RANGE	PROBABILITY	4	8	12	16
1-4	.3	8	12	16	20
5-8	.4	16	12	16	20
9-12	.2	24	24	16	20
13-16	.1	32	24	32	20
Avg.		16.8	15.6	17.6	20.0

Fig. 6.35. Numbers of bytes used for strings in various length ranges by different values of CPC.

Mass Storage of Character Strings

There is another approach to the storage of large numbers of character strings that combines the advantage of array storage (little overhead) with the advantages of linked-list storage (no wasted space due to padding, and no limit on string length). We create one very long array of characters, into which we shall store each character string. To tell where one string ends and the next begins, we need a special character called the *endmarker.* The endmarker character cannot appear as part of a legitimate character string. In what follows, we shall use * as the endmarker, for visibility, although it is more usual to choose a nonprinting character, such as the null character.

Endmarker

✦ **Example 6.18.** Suppose we declare an array `space` by

```
char space[MAX];
```

We can then store a word by giving a pointer to the first position of `space` devoted to that word. The `WORDCELL` structure, analogous to that of Example 6.16, would then be

```
typedef struct {
    char *word;
    int occurrences;
    LIST lines;
} WORDCELL;
```

In Fig. 6.36 we see the `WORDCELL` structure for the word `the` in a concordance based on the book of Genesis. The pointer `word` refers us to `space[3]`, where we see the beginning of the word `the`.

Note that the lowest elements of the array `space` might appear to contain the text itself. However, that would not continue to be the case for long. Even if the next elements contain the words `beginning`, `God`, and `created`, the second `the` would not appear again in the array `space`. Rather, that word would be accounted for by adding to the number of occurrences in the `WORDCELL` structure for `the`. As we proceeded through the book and found more repetitions of words, the entries in `space` would stop resembling the biblical text itself. ✦

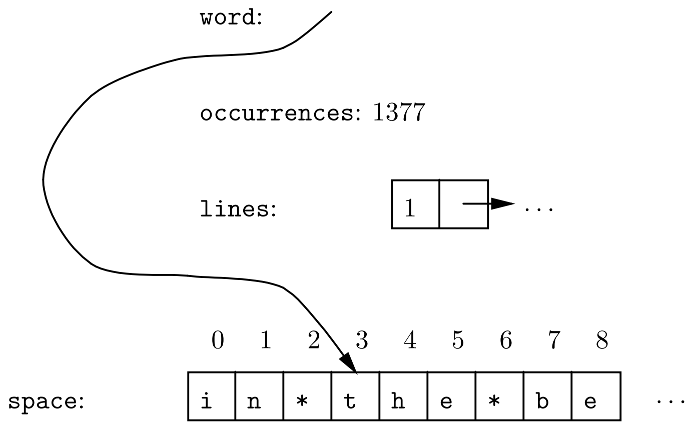


Fig. 6.36. Representing words by indices into string space.

As in Example 6.16, the structures of Example 6.18 can be formed into data structures such as binary search trees or linked lists by adding the appropriate pointer fields to the `WORDCELL` structure. The function $lt(W_1, W_2)$ that compares two `WORDCELL`'s W_1 and W_2 follows the `word` fields of these structures and compares them lexicographically.

To build a concordance using such a binary search tree, we maintain a pointer `available` to the first unoccupied position in the array `space`. Initially, `available` points to `space[0]`. Suppose we are scanning the text for which the concordance

What Happens When We Run Out of Space?

We have assumed that `space` is so large that there is always room to add a new word. Actually, each time we add a character we must be careful that the current position into which we write is less than `MAX`.

If we want to enter new words after running out of space, we need to be prepared to obtain new blocks of space when the old one runs out. Instead of creating just one array `space`, we can define a character-array type

```
typedef char SPACE[MAX];
```

We can then create a new array, the first character of which is pointed to by `available`, by

```
available = (char *) malloc(sizeof(SPACE));
```

It is useful to remember the end of this array by immediately assigning

```
last = available + MAX;
```

We then insert words into the array pointed to by `available`. If we can no longer fit words into this array, we call `malloc` to create another character array. Of course we must be careful not to write past the end of the array, and if we are presented with a string of length greater than MAX, there is no way we can store the word in this scheme.

is being built and we find the next word — say, `the`. We do not know whether or not `the` is already in the binary search tree. We thus temporarily add `the*` to the position indicated by `available` and the three following positions. We remember that the newly added word takes up 4 bytes.

Now we can search for the word `the` in the binary search tree. If found, we add 1 to its count of occurrences and insert the current line into the list of lines. If not found, we create a new node — which includes the fields of the `WORDCELL` structure, plus left- and right-child pointers (both `NULL`) — and insert it into the tree at the proper place. We set the `word` field in the new node to *available*, so that it refers to our copy of the word `the`. We set `occurrences` to 1 and create a list for the field `lines` consisting of only the current line of text. Finally, we must add 4 to `available`, since the word `the` has now been added permanently to the `space` array.

EXERCISES

6.10.1: For the structure type `WORDCELL` discussed in Example 6.16, write the following programs:

a) A function `create` that returns a pointer to a structure of type `WORDCELL`.

b) A function `insert(WORDCELL *pWC, int line)` that takes a pointer to the structure `WORDCELL` and a line number, adds 1 to the number of occurrences for that word, and adds that line to the list of lines if it is not already there.

6.10.2: Redo Example 6.17 under the assumption that any word length from 1 to 40 is equally likely; that is, 10% of the words are of length 1–4, 10% are of length 5–8, and so on, up to 10% in the range 37–40. What is the average number of bytes required if `CPC` is 4, 8, . . . , 40?

6.10.3*: If, in the model of Example 6.17, all word lengths from 1 to n are equally likely, what value of `CPC`, as a function of n, minimizes the number of bytes used? If you cannot get the exact answer, a big-oh approximation is useful.

6.10.4*: One advantage of using the structure of Example 6.18 is that one can share parts of the space array among two or more words. For example, the structure for the word `he` could have `word` field equal to 5 in the array of Fig. 6.36. Compress the words `all`, `call`, `man`, `mania`, `maniac`, `recall`, `two`, `woman` into as few elements of the `space` array as you can. How much space do you save by compression?

6.10.5*: Another approach to storing words is to eliminate the endmarker character from the `space` array. Instead, we add a `length` field to the `WORDCELL` structures of Example 6.18, to tell us how many characters from the first character, as indicated by the `word` field, are included in the word. Assuming that integers take four bytes, does this scheme save or cost space, compared with the scheme described in Example 6.18? What if integers could be stored in one byte?

6.10.6**: The scheme described in Exercise 6.10.5 also gives us opportunities to compress the `space` array. Now words can overlap even if neither is a suffix of the other. How many elements of the space array do you need to store the words in the list of Exercise 6.10.4, using the scheme of Exercise 6.10.5?

6.10.7: Write a program to take two `WORDCELL`'s as discussed in Example 6.18 and determine which one's word precedes the other in lexicographic order. Recall that words are terminated by `*` in this example.

✦ 6.11 Summary of Chapter 6

The following points were covered in Chapter 6.

✦ Lists are an important data model representing sequences of elements.

✦ Linked lists and arrays are two data structures that can be used to implement lists.

✦ Lists are a simple implementation of dictionaries, but their efficiency does not compare with that of the binary search tree of Chapter 5 or the hash table to be covered in Chapter 7.

✦ Placing a "sentinel" at the end of an array to make sure we find the element we are seeking is a useful efficiency improver.

✦ Stacks and queues are important special kinds of lists.

✦ The stack is used "behind the scenes" to implement recursive functions.

- A character string is an important special case of a list, and we have a number of special data structures for representing character strings efficiently. These include linked lists that hold several characters per cell and large arrays shared by many character strings.
- The problem of finding longest common subsequences can be solved efficiently by a technique known as "dynamic programming," in which we fill a table of information in the proper order.

6.12 Bibliographic Notes for Chapter 6

Knuth [1968] is still the fundamental source on list data structures. While it is hard to trace the origins of very basic notions such as "list" or "stack," the first programming language to use lists as a part of its data model was IPL-V (Newell et al. [1961]), although among the early list-processing languages, only Lisp (McCarthy et al. [1962]) survives among the currently important languages. Lisp, by the way, stands for "LISt Processing."

The use of stacks in run-time implementation of recursive programs is discussed in more detail in Aho, Sethi, and Ullman [1986].

The longest-common-subsequence algorithm described in Section 6.9 is by Wagner and Fischer [1975]. The algorithm actually used in the UNIX `diff` command is described in Hunt and Szymanski [1977]. Aho [1990] surveys a number of algorithms involving the matching of character strings.

Dynamic programming as an abstract technique was described by Bellman [1957]. Aho, Hopcroft, and Ullman [1983] give a number of examples of algorithms using dynamic programming.

Aho, A. V. [1990]. "Algorithms for finding patterns in strings," in *Handbook of Theoretical Computer Science* Vol. A: *Algorithms and Complexity* (J. Van Leeuwen, ed.), MIT Press, Cambridge, Mass.

Aho, A. V., J. E. Hopcroft, and J. D. Ullman [1983]. *Data Structures and Algorithms*, Addison-Wesley, Reading, Mass.

Aho, A. V., R. Sethi, and J. D. Ullman [1986]. *Compilers: Principles, Techniques, and Tools*, Addison-Wesley, Reading, Mass.

Bellman, R. E. [1957]. *Dynamic Programming*, Princeton University Press, Princeton, NJ.

Hunt, J. W. and T. G. Szymanski [1977]. "A fast algorithm for computing longest common subsequences," *Comm. ACM* **20**:5, pp. 350–353.

Knuth, D. E. [1968]. *The Art of Computer Programming*, Vol. I, *Fundamental Algorithms*, Addison-Wesley, Reading, Mass.

McCarthy, J. et al. [1962]. *LISP 1.5 Programmer's Manual*, MIT Computation Center and Research Laboratory of Electronics, Cambridge, Mass.

Newell, A., F. M. Tonge, E. A. Feigenbaum, B. F. Green, and G. H. Mealy [1961]. *Information Processing Language-V Manual*, Prentice-Hall, Englewood Cliffs, New Jersey.

Wagner, R. A. and M. J. Fischer [1975]. "The string to string correction problem," *J. ACM* **21**:1, pp. 168–173.